AD-A110 866 INSTITUTE FOR DEFENSE ANALYSES ARLINGTON VA SCIENCE A--ETC F/G 9/2
REPORT OF IDA SUMMER STUDY ON HARDWARE DESCRIPTION LANGUAGE.(U)
OCT 81 G W PRESTON MDA903-79-C-0202
UNCLASSIFIED IDA-P-1595 IDA/HQ-81-23681 NL

1 of 2
AD
A110866

Copy 13 of 120 copies

AD A110866

2 LEVEL II

/62p

IDA PAPER P-1595

REPORT OF IDA SUMMER STUDY ON HARDWARE DESCRIPTION LANGUAGE

DTIC FILE COPY

C. W. Preston

DTIC
ELECTE
FEB 12 1982
S D
B

October 1981

Prepared for

Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION

403 115

82 02 09 095 IDA ~~Log No.~~ HQ 81-23681

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A110 866	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Report of IDA Summer Study on Hardware Description Language		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER IDA Paper P-1595
7. AUTHOR(s) G. W. Preston		8. CONTRACT OR GRANT NUMBER(s) MDA 903 79 C 0202
9. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Defense Analyses 400 Army-Navy Drive Arlington, VA 22202		10. PROGRAM ELEMENT PROJECT TASK AREA & WORK UNIT NUMBERS Task T-184
11. CONTROLLING OFFICE NAME AND ADDRESS DUSD (Research and Advanced Technology) The Pentagon, Washington, DC 20301		12. REPORT DATE October 1981
		13. NUMBER OF PAGES 153
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE NA

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

None

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Hardware Description Language (HDL), Computer Aided Design (CAD), Very High Speed Integrated Circuit (VHSIC), Register Transfer Language (RTL), hierarchical language, technology insertion.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Institute for Defense Analyses' Summer Study on Hardware Description Language met from June 1 through June 12, 1981, for the purpose of determining the goals and requirements for a VHSIC-level hardware description language (VHDL).

Three major results were accomplished. First, the behavioral, structural hierarchical, and other requirements that such a language would need to fulfill to meet DoD VHSIC were detailed. Second, the existing Texas

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Contd.)

Instruments HDL was examined and a group of changes and additions were recommended (resulting in "WHDL") which meets many of the requirements of VHDL. Third, determination was made of those features of Ada which would be needed for VHDL and additional constructs (outside of Ada) which would be required.

IDA PAPER P-1595

REPORT OF IDA SUMMER STUDY ON HARDWARE DESCRIPTION LANGUAGE

G. W. Preston

October 1981

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
400 Army-Navy Drive, Arlington, Virginia 22202
Contract MDA 903 79 C 0202
Task T-184

PREFACE

The purposes of this study could not well be explained without reference to its historical context. The name hardware description language (HDL) itself has undergone changes in connotation which might well perplex readers of this document, even those who contributed to the early development of HDLs. As a matter of fact, in the context of the VHSIC (Very High Speed Integrated Circuit) program, the scope and purpose of the hardware description language has been enlarged to such an extent that the HDL (in its original meaning) can be properly seen as only one thread in the history of the technology encompassed by the (as yet unimplemented) language contemplated by this study. It is for this reason that we introduce the acronym VHDL (for VHSIC HDL).

This new language (VHDL) also draws its concepts from:

(1) modern higher-order programming languages, Ada in particular (which is structured, modular, provides for user-defined data objects, etc.)

(2) the discipline and procedures of very-large-scale integrated circuit (VLSI) design (again, structured and modular)

(3) languages for system performance specifications, testing, and simulation

(4) the techniques of axiomatic system description and design verification and validation.

This effort falls squarely within the tradition of rationalized techniques for managing complex human endeavors, perhaps the most significant outcome of the computer revolution, which was virtually unforeseen by its pioneers In this case, these techniques are being brought to bear upon one of the more intricate processes ever undertaken--the design of microelectronic circuits consisting of hundreds of thousands of active elements.

But the HDL, even for its original narrower purpose, would require considerable restructuring at this time because of another outgrowth of integrated circuit technology, namely hardware functionality (the use of dedicated blocks of circuitry for special complex purposes, the hardware macro). This development contains the seeds of an eventual reconciliation between the hardware and software thinker and an end to the tedium of microcode.

The evolution of HDL reflects a slow and painful adaption to the necessity for expressing intent rigorously, in modular form, and, above all, hierarchically. Perhaps the original purpose of hierarchical description was to "divide and conquer" by describing designs in a sufficiently compact, abstract way as to be comprehensible--<u>in their entirety</u>; then, at successively lower levels of abstraction, to introduce detail (algorithmic on the behavioral side, implementational on the structural); terminating finally in details of the most primitive elements. At each successive level conformity to the higher level can be verified and validated; ultimately through detailed simulation from the primitive level upward. VHDL encompasses hierarchical hardware description (behavioral and functional) from the VHSIC brassboard level to the hardware primitive.

As it turns out, the hierarchical language provides another equally important capability--namely adaptability to progress in technology. The ongoing revolution in integrated circuit technology has, of necessity, created a community with a high level of tolerance to "future shock"--a community which seems to stand with one foot in this century and one in the next. Part of this expertise addresses the cost of progress, e.g., rapid product obsolescence, hence high research and development budgets relative to sales and product expansion limited by human resources in the form of design teams. Nowhere is the cost of progress more extreme than in military systems. Hierarchical HDL helps mitigate this high cost of progress by permitting insertion of new technology (algorithmic or structural) without disturbing the description above the level of the insertion. In other words, the hierarchical language is structured to adapt to evolution.

It was Irving Reed who (in 1952) introduced the term Register Transfer Language (RTL) in connection with computer design. This concept fell somewhere between a structural and behavioral description of computer design. Seymour Cray also developed some ideas on computer descriptions in the 1950s based upon Boolean equations. The direct lineal descendants of these languages are Computer Hardware Description Languages (CHDLs) (e.g., AHPL, ISPS, CDL, DDL), pursued principally in universities. These CHDLs primarily addressed the machine (rather than chip) level of design.

With the advent of VLSI in the 1970s, computer-aided design (CAD) became a necessity instead of a laboratory curiosity and, in industry, design languages sprang up to assist the designer at each level of design in communicating with the various CAD tools. The Department of Defense (DoD) perceived the need to make this kind of communication available as a standard way for various contractors to share design data and design efforts. However, contractors were (and are) reluctant to share freely design tools that were developed at considerable company expense. Proprietary design tools were an impediment to DoD accomplishment of its own goals. The design languages of each of these companies were and are closely tied to their data bases and design tools.

In order to circumvent these problems, the suggestion was made that a broad-based standard HDL be created which would allow transfer of design descriptions and design data independent of any given design data base or design tool, yet be machine-readable--a data-transfer language and a standard machine-readable documentation language.

As a first step in the development of such an HDL, the Institute for Defense Analyses (IDA) organized a meeting in June 1980 to bring together the nine VHSIC contractors, some university researchers in CHDLs, and DoD representatives. The recommendations that sprang from this meeting led to the creation of a tri-Service/industry committee to develop the standard DoD HDL. One committee meeting was held (October 1980) to plan meaningful development activities for HDL goals. These plans were not fulfilled; first, because of contention for the time of the committee members at their home bases (VHSIC proposal efforts), and, second, by the length of the procurement process itself.

In the spring of 1981 IDA organized this study.

ACKNOWLEDGMENTS

The author of this document is, quite literally, a group--an ad hoc assembly which, for two weeks, functioned at a level of dedication, enthusiasm, and selflessness that could not easily be imagined if not seen. It in no way qualifies that acknowledgment to single out Ron Waxman and Dan Nash for their untiring leadership, and Anthea DeVaughan for her superb staff support.

EXECUTIVE SUMMARY

The Institute for Defense Analyses was requested by OUSDR&E, in the fall of 1980, to undertake a study, the objective of which was "to contribute to the development of a standard hardware description language (HDL) for the Very High Speed Integrated Circuit (VHSIC) program." In partial fulfillment of this objective, a summer study was organized at the National Academy of Sciences Study Center, Woods Hole, Massachusetts, from June 1 through June 12, 1981, to develop specifications for a new hardware description language for VHSIC. Thirty-four specialists from fifteen corporations (engaged in the development and manufacture of computers, integrated circuits, military systems, aerospace systems, etc.), five universities, four Federal Contract Research Centers, and one non-profit research institute participated. This approach brought to bear upon the problems (of developing such a language) the talents of a considerable representation of the world's foremost experts in this field, and also advanced the establishment of a consensus which would be essential if the proposed standards were to be accepted.

The report of the group's accomplishment was subsequently refined through individual efforts (notably members of the WHDL Committee--Appendix E) and at a final meeting of the committee (attended by over half the participants in the Summer Study) at IDA on Tuesday, September 22, 1981, to review the final draft.

The new language (designated VHDL for VHSIC HDL) addresses a group of critical issues relating to military applications

of VHSIC technology. In a narrow sense, the purposes of VHDL relate to:

- reducing the cost and schedule for integrated circuit design
- the documentation of integrated circuit designs for systems designers and other users
- the transfer of design data for purposes of subcontracting, second sourcing, etc.
- technology upgrading (semiconductor or algorithmic).

The more general purposes of VHDL relate to:

- specification of military systems design and performance
- insertion of VHSIC chips into military equipment
- maintaining operational readiness through a reduction in out-year logistics failure
- future upgrades in system performance through new technology insertion.

The Summer Study developed a list of recommended specifications pertaining to the hierarchical nature and the behavioral and structural features of VHDL. In addition, an existing HDL (the Texas Instruments HDL) was examined and a detailed set of modifications and extensions were recommended which would meet most of the VHDL specifications. This (possibly interim) language was designated WHDL. Finally, a determination was made of those features of Ada which would be applicable to an entirely new language based on the VHDL specifications and of additional features and constructs outside of Ada which would have to be introduced.

The main body of the report details the VHDL specifications. Appendix A describes, in some detail, an alternative language, WHDL, which is derived from the Texas Instruments HDL and incorporates most of the features of VHDL (the addendum on behavioral concurrency is noteworthy). Appendix B discusses the applicability of Ada concepts to VHDL.

All participants were invited to write their personal comments on the work of the Summer Study. These are contained in Appendix C in their entirety and comprise an important body of commentary.

ABSTRACT

The Institute for Defense Analyses' Summer Study on Hardware Description Language met from June 1 through June 12, 1981, for the purpose of determining the goals and requirements for a VHSIC-level hardware description language (VHDL).

Three major results were accomplished. First, the behavioral, structural, hierarchical, and other requirements that such a language would need to fulfill to meet DoD VHSIC were detailed. Second, the existing Texas Instruments HDL was examined and a group of changes and additions were recommended (resulting in "WHDL") which meets many of the requirements of VHDL. Third, determination was made of those features of Ada which would be needed for VHDL and additional constructs (outside of Ada) which would be required.

CONTENTS

	Preface	iii
	Acknowledgments	vii
	Executive Summary	ix
	Abstract	xi
1.0	INTRODUCTION	1-1
1.1	Purposes of VHDL	1-2
1.2	Documentation	1-2
1.3	High-Level Design	1-5
1.4	Use by Design Automation Tools	1-6
1.4.1	Use by Simulation Tools	1-6
1.4.2	Use by Synthesis Tools	1-6
1.4.3	Use by Software Tool Generation Programs	1-6
1.4.4	Use by Testing Tools	1-6
1.4.5	Use by Physical Design Tools	1-7
1.5	Translation to Other HDLs	1-7
1.6	Portability	1-8
1.7	Application	1-8
2.0	HIERARCHY CONSIDERATIONS	2-1
2.1	Reference of Objects	2-5
2.2	Collections of Related Objects	2-6
2.3	Modularity	2-7
2.4	Libraries	2-7
2.5	Completeness	2-7
2.6	Recursion	2-8
3.0	BEHAVIOR	3-1
3.1	Behavioral Descriptions	3-1
3.2	Data Types	3-2
3.2.1	Data Abstraction	3-2
3.2.2	Enumeration	3-3
3.2.3	Range	3-3
3.2.4	Precision	3-3
3.2.5	User-defined Data Types	3-3
3.2.6	Deferred Definition	3-5
3.2.7	Strong Type Checking	3-6
3.2.8	Coercion	3-6
3.2.9	Reference to Attributes	3-7
3.2.10	Scope and Visibility	3-7
3.3	Operators and Expressions	3-8

3.3.1	Basic Operators	3-8
3.3.2	Compare Under MASK	3-9
3.3.3	User-Defined Operators	3-9
3.4	Control Flow	3-9
3.4.1	Control Constructs	3-10
3.4.2	Control Abstraction	3-10
3.5	Timing	3-10
3.5.1	Synchronous and Asynchronous Time	3-11
3.5.2	Simulation Option Interfaces	3-11
3.5.3	Timing Constraints	3-11
3.5.4	Parallel Element Models	3-12
3.5.5	Rise and Fall Times	3-13
3.5.6	Clock Definitions	3-13
3.5.7	Time Control Concepts	3-13
3.5.8	Termination of Currently Executing Procedures	3-14
4.0	STRUCTURE	4-1
4.1	Connections	4-1
4.2	Generic Models	4-1
4.2.1	Purpose	4-1
4.2.2	Parts	4-2
4.2.3	Communication	4-2
4.2.4	Parameters	4-3
4.2.5	Declaration Scope	4-3
4.2.6	Behavior	4-3
4.2.7	Structure	4-3
4.2.8	Instance Names	4-4
4.3	Permutability	4-4
4.4	Generic Instantiation	4-5
4.5	Iteration in Structure Description	4-5
4.6	Procedural Modeling	4-8
4.7	Extensible Number of Primitive Components	4-8
5.0	GUIDELINES	5-1
5.1	Kernel Plus Extensions Equals Language	5-1
5.2	High-Level Parallelism	5-3
5.3	Verifiability	5-3
5.4	One Language	5-4
5.5	Technology Independence	5-4
5.6	Meaningful Key Words	5-4
5.7	Comments	5-5
5.8	Compiler Directives	5-5
5.9	Expression of Invariance	5-5
5.10	LR(1) Grammar	5-5
5.11	Execution Conflicts	5-6
5.12	Multi-Valued Logic	5-7
5.13	Incremental Compilation	5-7
5.14	Exception Handling	5-7

APPENDIX A--WHDL A-1

APPENDIX B--Applicability of Ada Concepts to HDL B-1

APPENDIX C--Individual Comments C-1

APPENDIX D--Speakers and Presentations D-1

APPENDIX E--Participants in IDA Summer Study on Hardware Description Language E-1

APPENDIX F--Glossary F-1

1.0 INTRODUCTION

This report documents the work of the Institute for Defense Analyses' Summer Study on Hardware Description Language (HDL) held June 1-12, 1981, at the National Academy of Sciences' Study Center, Woods Hole, Massachusetts. The study was organized by IDA under OUSDRE sponsorship in cooperation with the Very High Speed Integrated Circuit (VHSIC) Tri-Service HDL Committee, which will formally review its recommendations. There were thirty-four participants representing fifteen companies (IC manufacturers, systems suppliers, computer manufacturers), four Federal Contract Research Centers, one non-profit organization, five universities, and the Air Force (see Appendix E).

The purpose of the study was to establish consensus specifications for a DoD standard HDL, particularly for use in the ongoing VHSIC program. Because the required language is to provide capabilities that go far beyond earlier HDLs, the acronym VHDL (for VHSIC HDL) is introduced. In preparation for this task, the group reviewed the Sperry-Univac "VHSIC HDL: Requirements Report" and the Texas Instruments' HDL (which was graciously contributed by that corporation as a STRAWMAN HDL) and heard 13 technical presentations, many of which dealt with new and innovative work (Appendix D).

The main body of this report documents the recommendations of the entire group for facilities and features of VHDL.

A subcommittee prepared a subset of VHDL (WHDL), described in Appendix A, which incorporates many VHDL features into the STRAWMAN. A second subcommittee evaluated the use of Ada, listing unneeded features within Ada and necessary extensions for the VHDL (Appendix B).

1.1 PURPOSES OF VHDL

This section defines the purposes for which the VHDL will be used. It is intended that the design of the language be oriented to fulfill the end purposes illustrated in Table 1-1.

The VHDL would provide the means to efficiently describe VHSIC integrated circuits and the digital logic portions of the system in which the chips are interconnected, at least to the VHSIC "brassboard" level. The VHSIC chips consist of about 30,000-100,000 equivalent gates. The systems in which they will be included may be networks of up to several million equivalent gates.

In toto, VHDL must eventually serve the purposes of:

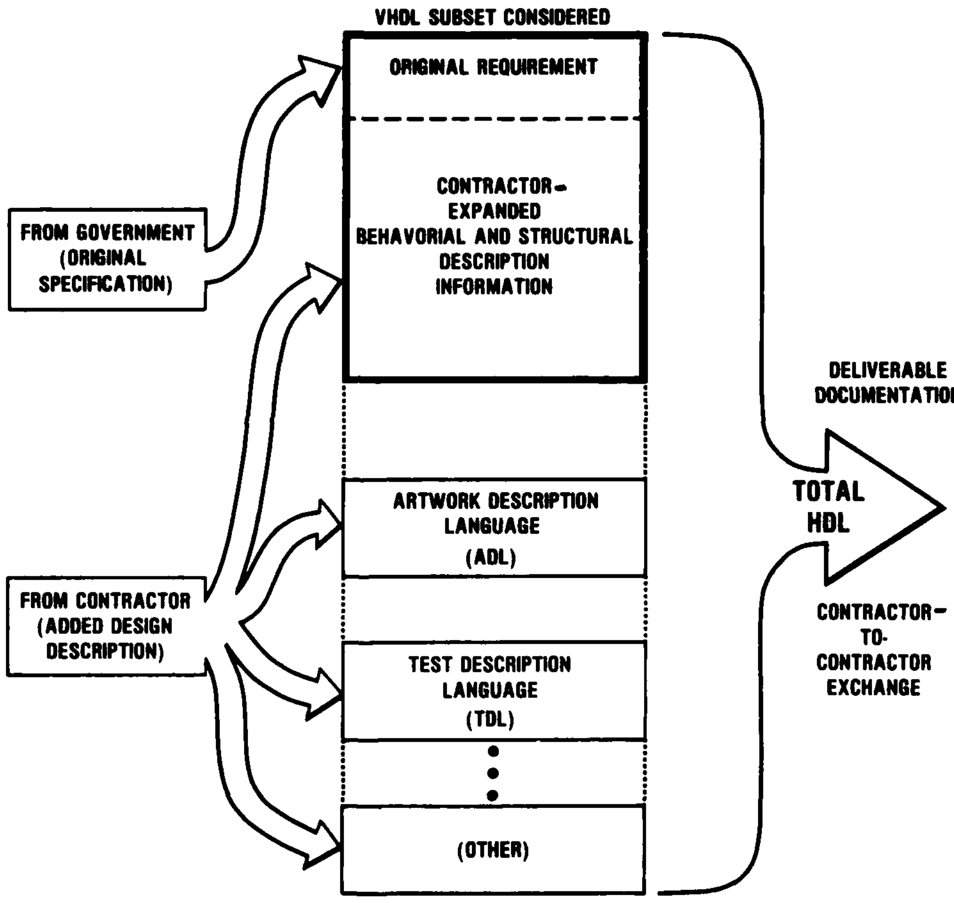
- systems specification
- circuit design
- system development
- system upgrades
- logistics support
- technology upgrades.

At the system level, designs of unprecedented complexity must perform their intended functions, in some cases, with no acceptable margin for undetected error. Even extremely rare potential pathological modes of operation must be prevented. This demands hardware description of the highest possible degree of rigor and completeness. On the other hand, within these constraints, competition and technology innovation must be allowed the greatest freedom. However, only a limited subset of these purposes was directly addressed by the study group.

1.2 DOCUMENTATION

The VHDL should include as a minimum all of the attributes listed in Table 1-2 as they are required for complete documentation. The VHDL subset being addressed in this document is limited to the I/O Interface, Function (Behavior), Structure,

TABLE 1-1. VHDL USE FOR TWO-WAY DESIGN DOCUMENTATION AND TRANSFER


VHDL SUBSET CONSIDERED
ORIGINAL REQUIREMENT
CONTRACTOR-EXPANDED BEHAVORIAL AND STRUCTURAL DESCRIPTION INFORMATION
FROM GOVERNMENT (ORIGINAL SPECIFICATION)
FROM CONTRACTOR (ADDED DESIGN DESCRIPTION)
ARTWORK DESCRIPTION LANGUAGE (ADL)
TEST DESCRIPTION LANGUAGE (TDL)
(OTHER)
DELIVERABLE DOCUMENTATION
TOTAL HDL
CONTRACTOR-TO-CONTRACTOR EXCHANGE

7-23-81-6

TABLE 1-2

HDL ATTRIBUTE	CHARACTERISTICS AND CONTENT
Function (Behavior)	Description (e.g., 32-bit multiply, characteristic/mantissa, result) Behavior (I/O transform) Information flow (algorithm) Specification of timing constraints and dependencies Functional test information
Structure	Hierarchical description of primitives and their interconnection
Physical Boundary	Package shape Pin placement Layout and boundary description
I/O Interface	Connector/package type Pin/signal list
Electrical Boundary Characteristics	Signals Voltages Drive Load
Timing	Timing constraints on I/O (internal timings are carried with the behavioral blocks)
Physical environment	Temperature Vibration Humidity Radiation Power
Test	Test language description of test vector set for the given entity (e.g., chip macro, etc.)
Artwork	Required at chip level, board level, etc.
Configuration Control Block	Revision history Version history Configuration management
Design constraints	Reliability Maintainability Testability Diagnostic isolation

and Timing attributes. Within this subset, documentation features that may be included are listed below. Actual content is dependent upon each specific design.

The VHDL subset may include all of the documentation features listed below:

- I/O interface
- I/O behavior of the design at the primary inputs and outputs--regardless of internal construction and regardless of timing
- I/O behavior of the design at the primary inputs and outputs--with respect to time and time-related constraints and parameters
- behavioral description at each level of the design
- structural decomposition of the design into logical and/or physical entities
- all necessary information to fulfill textual requirements of MIL-M-38510
- memory contents necessary to implement machine behavior
- functional test description.

The actual contents will be dependent upon agreement among contractors and/or government on a per-design basis.

The VHDL shall be deliverable in both machine-readable ASCII character set form (with upper and lower cases considered equivalent) and in textual form.

1.3 HIGH-LEVEL DESIGN

The VHDL shall be usable both as a user-oriented design language and a deliverable documentation standard. The specific recommendations made in succeeding chapters will be oriented toward producing a design language as a part of VHDL.

To convert existing simulation or other CAD tools to operate from VHDL, operation of existing design tools could continue with generation of user translators to permit conversion of the design data to or from the VHDL.

1.4 USE BY DESIGN AUTOMATION TOOLS

Nothing shall be done in the VHDL in the light of existing knowledge to preclude the future extension of the language to express new concepts necessary to support future design automation tools.

1.4.1 Use By Simulation Tools

The VHDL code should provide sufficient information to allow verification of the design by simulation or equivalent tools. Specific levels of simulation to be supported are functional, register-transfer, and gate level. It is recognized that such levels are arbitrarily defined and may not be used by all contractors.

1.4.2 Use by Synthesis Tools

The VHDL should be designed with a view toward automatic synthesis.

1.4.3 Use By Software Tool Generation Programs

The utility of VHSIC hardware will depend on the ability to produce software to execute on the hardware systems. It will be necessary to have a collection of support software programs to aid in the programming of the VHSIC hardware such as compilers, assemblers, and instruction-level simulators (see Fig. 1-1). VHDL should be rich enough to be used as input to a table-building program which supports table-driven code generation for compilers. For assemblers, one possibility is the use of the enumeration type, with the enumeration type representation specification of Ada to map an assembly language specification to the required machine language.

1.4.4 Use by Testing Tools

The VHDL code should provide sufficient information to support the development of hardware tests. As a minimum, hardware test development shall be defined as the development of a set of vectors to detect single stuck-at-one and single stuck-at-zero faults at a user-definable circuit node.

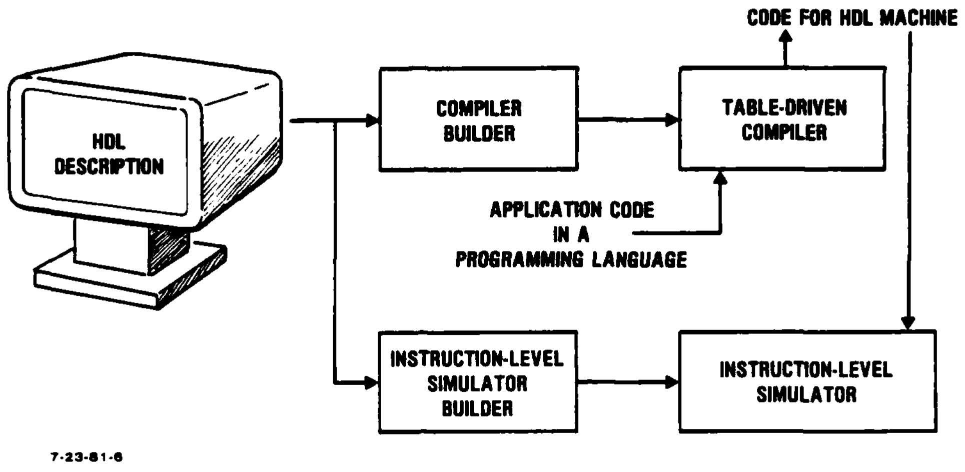

CODE FOR HDL MACHINE
HDL DESCRIPTION
COMPILER BUILDER
TABLE-DRIVEN COMPILER
APPLICATION CODE IN A PROGRAMMING LANGUAGE
INSTRUCTION-LEVEL SIMULATOR BUILDER
INSTRUCTION-LEVEL SIMULATOR
7-23-81-6

FIGURE 1-1. Application code in a programming language

1.4.5 Use by Physical Design Tools

The VHDL code should provide sufficient information to support the physical design of a VHSIC system. Minimum information would include a list of physical components and a description of the interconnection of such components. The VHDL should provide a format that allows topological placement and/or geometric information at a level of detail sufficient to facilitate designer-influenced physical design.

1.5 TRANSLATION TO OTHER HDLs

No legal restrictions on the VHDL should preclude its translation into another HDL.

Variations in users' CAD systems, styles, and indigenous HDLs shall be the responsibility of the individual users, and accommodated by locally applied translators and/or application disciplines.

1.6 PORTABILITY

Since VHDL is to be used as a means of transmitting design data between contractors, it is required that the language be portable. Portability requires that VHDL data be deliverable in both machine-readable ASCII character set form (with upper and lower case considered equivalent) and in textual form. The user may have the option of how to use the data in the design process.

It is envisioned that a major use of the VHDL (for contractors with design systems) will be to drive design tools. Thus, the VHDL may be compilable to object languages that execute on various data-processing machines. The object language will, of course, be machine-dependent. At some point, standard compilers may be created which compile the VHDL into object code that can drive public-domain design tools. Individual contractors may also have their own internal HDLs in source or object form. Translation from (to) the VHDL to (from) a contractor's internal design language is the responsibility of that contractor and may be performed, at the contractor's option, on the source HDL or on the internal object code.

1.7 APPLICATION

The VHDL specification document should replace the configuration item specification and associated technical description documents, i.e., it should substitute for, rather than add to, current deliverable documentation.

2.0 HIERARCHY CONSIDERATIONS

The VHDL must support the description of a hierarchical representation of the hardware.* Each level of VHDL design model contains design data, including I/O interface, structure, behavior, and environmental constraints (including test cases and expected results). The model consists of a hierarchy of design entities, each of which may be further decomposed into its own constituent components. These components may be defined as design entities for further decomposition. More than one description or decomposition may be used to describe a given design entity. The behavior of a design entity is considered to be one of these alternative descriptions, with the stipulation that its internal organization need not correspond to the actual hardware decomposition. One may refer to such a behavior as a logical (or functional) description of the design entity. A behavior at any given level could be constructed by combining lower-level behaviors. A decomposition may lead to a purely physical package of a design. All alternative decompositions must be I/O equivalent, i.e., they are functionally interchangeable. An illustration of the hierarchical model is shown in Fig. 2-1.

Figure 2-2 illustrates one breakdown of the different levels of abstraction involved in the design process. The mathematical algorithm plus a choice of design style (e.g., serial vs. parallel, register transfer vs. data flow, clocked vs. self-timed) results in a machine architecture. From there, a choice of an implementation technique (e.g., a structured logic layout technique vs. random logic) leads to a design description

*see Appendix C, p. C-29

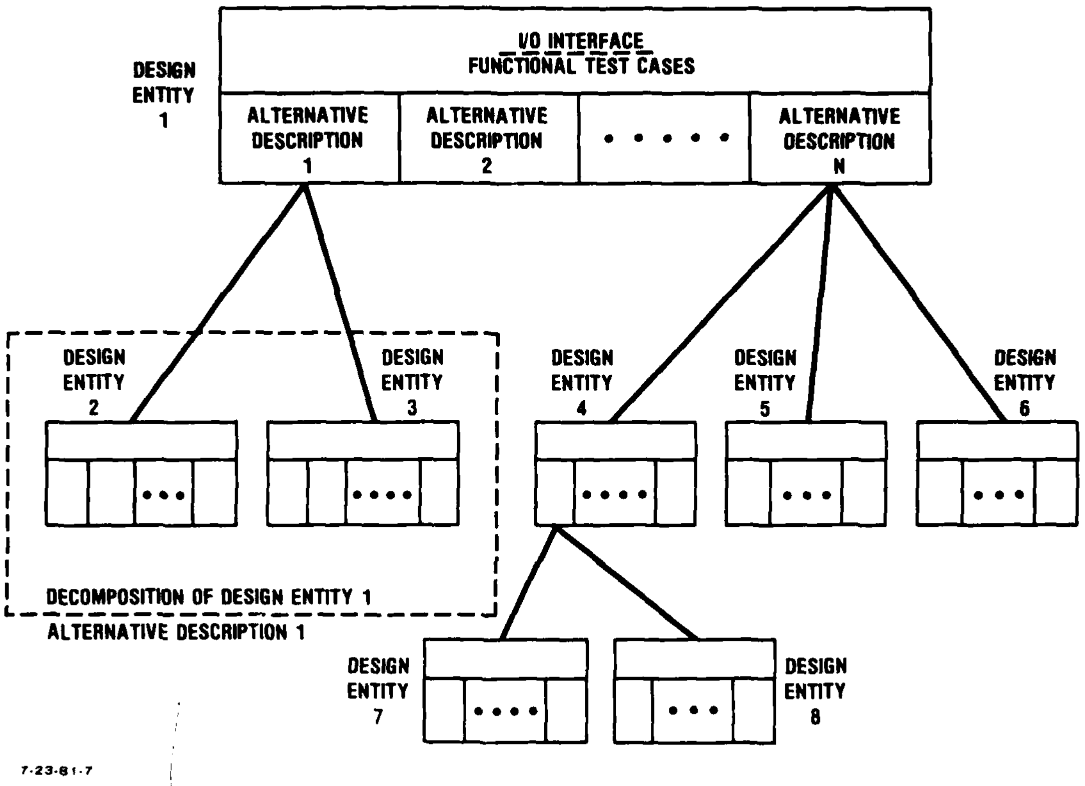


FIGURE 2-1. Design entities, decomposition, and alternative descriptions

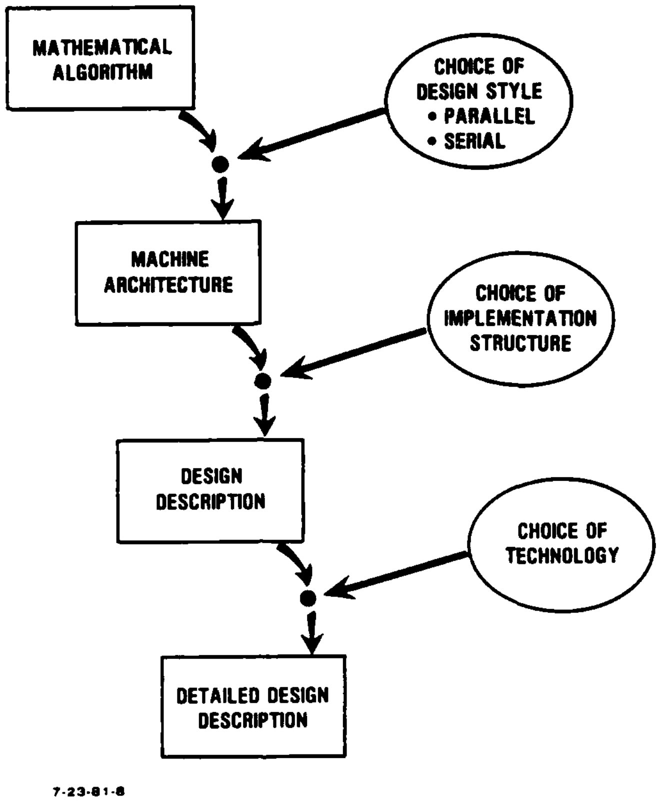


FIGURE 2-2. Levels of abstraction

(e.g., PLA truth tables, SLA programs, etc.). A choice of technology then leads to a detailed design description.

Specific characteristics of the design model are:

- I/O signals should be specified at each level of description.
- All design entities except primitive components are described as a structure of lower level design entities that includes a description of their

interaction, including the accommodation of parameter passing.

- The logical interaction of components is described as an exchange of signals, including the timing of the occurrence of changes in signal levels.
- The behavior of all lowest-level components must be completely described in VHDL.
- Timing data should be specifiable at each level of description, allowing for form and accuracy which are appropriate to the specific design entity.
- Each level of hierarchical decomposition may be composed of I/O equivalent alternative sets of behaviors and/or further decompositions.

An example of a four-level hierarchical decomposition of a microprocessor is shown in Figure 2-3. In this example, "A" is a primitive because it has no structural decomposition.

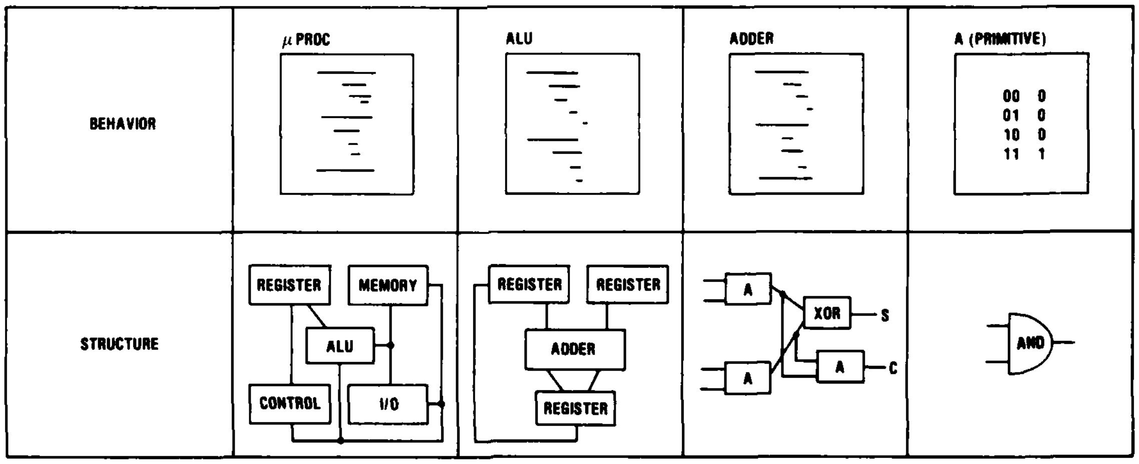


FIGURE 2-3. Hierarchical decomposition of a microprocessor

The language should contain some construct to permit definition of new design entities (as part of the design entity being defined) or include definitions from a library. With such a mechanism, basic design entities (e.g., for gates or loops) can be defined and stored in a library. Each designer can then use whichever design entity is most appropriate, or define his or her own. With this approach there is no need for the language to have built-in design entities.

For any design, there is a lowest level of component that is used. Components at this level are called primitive components. To fully document or transport a design, each primitive component must have at least an interface specification and a behavioral description in HDL.

At each level of the hierarchical description, i.e., for each design entity, there should be a description of the design entity composed of:

- I/O interface specification
- a behavioral desription: transfer functions between I/O ports
- a logical or physical structural description expressing the interconnection of components to achieve the described behavior
- a list of required performance parameters: timing characteristics, precision, repetition rate, etc. (optional below the system level).

2.1 REFERENCE OF OBJECTS

Means should be provided to clearly identify every instance of a generic design entity. Identification is to be automatically related to where the instance appears within the hierarchy as well as within a level (first instance...last instance), for instances that are generated as the result of a variable index.

Examples of this and other naming problems are:

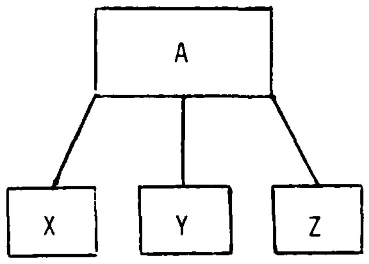
- Design entities that come from generic design-entity instantiation

- Components that come from design-entity instantiation
- Fields of a record
- Elements of an array
- Attributes of a design entity
- Terminals of a design entity
- Networks of a design entity.

All the above potential naming problems should be resolved so that in documentation, simulation, or test vector generation each of these objects will be uniquely identified.

2.2 COLLECTIONS OF RELATED OBJECTS

Language constructs are needed in HDL to enable the collection, organization, and naming of HDL descriptions and to define the relationships between these descriptions. In a hierarchical description, a means must be provided to indicate that a node is decomposed into other nodes. Thus, in the example below, if node A is decomposed into X, Y, and Z, then a means must exist in the language to describe the relationships among A, X, Y, and Z.



HDL descriptions may be applicable to more than one design, or may be multiply used within a given design. Naming conventions must allow for these situations.

The ability to decompose an HDL description into lower-level HDL descriptions requires the existence of organization and relational constructs.

The need to collect various HDL descriptions to fulfill the function of a higher-level HDL description requires collection constructs in addition to naming, organization, and relational constructs.

2.3 MODULARITY

The division of structure and behavior into sections is an important aspect of system specification and design. The HDL should have syntax that permits effective description of logical and/or physical modularity and interconnection of components. Modularity should be consistent and traceable in hierarchical descriptions.

2.4 LIBRARIES

The language must permit libraries at all levels as long as the items in the libraries are described in the standard HDL syntax. The levels may extend down to the cell level to permit complete structural description in the HDL. For example, cell instantiation should be retained in the HDL to permit the use of hierarchical aids during processing steps (E-beam exposure, etc.).

2.5 COMPLETENESS

The VHSIC HDL should allow for the complete description of hardware. To properly document (and potentially transfer) a given design, each of its design entities must be represented by its complete I/O interface and one of its alternative descriptions. If any reference is made to a library description, such description is considered part of the design.

Another aspect of completeness relates to the assumption that all alternative descriptions of a design entity are I/O equivalent (in a functional sense). The problems associated with establishing this are the same as those faced by software verification generally.

2.6 RECURSION

Some problems are more suited to recursive description than others. Mechanisms for recursive behavior and structure descriptions should be provided for this reason. Care should be exercised in designing the language constructs for expressing behavior and structure descriptions to ensure that infinite recursion is avoided or detected. Recursion need not be supported in the first release of the VHDL compiler.

3.0 BEHAVIOR

The behavior section of a design entity must be either totally procedural (sequentially executed programming language) or totally non-procedural as in a Register Transfer Language. The behavior section should contain data declarations, program control statements, and expressions detailing the transformations expected in the system described. The data declarations identify the various signals and memory variables useful in determining the course of events described by the behavior. Iteration, selection, and branching options should allow the user to control the path of execution taken by the program. Operations on operands are combined in various sequences to produce the values determined from expression evaluation and replacement. Output values are then evaluated and transferred to other parts of the described system.

3.1 BEHAVIORAL DESCRIPTIONS

Procedurally-oriented behavioral descriptions should be similar to concepts found in programming languages--the subroutine from FORTRAN or the procedure in Pascal. They may receive inputs from other parts of the system, store previous signal values, pass data items among themselves, and influence signals that are propagated to the rest of the system as outputs. The behavioral descriptions may be executed in sequence, in parallel, or in some combination of paths.

Behavioral descriptions should be composed of two parts--the declaration section and the execution section. The declaration section identifies the local variables of interest to the procedural description. The signals used to communicate to the other parts of the system are identified and defined in the I/O interface specification. The execution section is composed of program control statements and expressions detailing the data transformations expected in the system described. The behavioral description may be hierarchical, reflecting levels of control and data abstraction.

3.2 DATA TYPES

The user should have the ability to define data objects that can be used to represent a wide variety of possible meanings. The built-in data types should be patterned after Ada and should include:

- enumeration types
- integer types
- REAL type.

Options should exist for identifying the range of values that are permissible. Larger groupings or associations of multiple objects should be able to be declared to handle the possibilities of records and records with field variants.

The user should be able to specify the precision of data objects at all levels. Type checking should be accommodated with the provision for controlled override. Constructs should be provided for accessing compiler-defined constants for data storage attributes (as in Ada).

3.2.1 Data Abstraction

Data abstraction includes both the concepts of grouping "fields" into "records" and of defining higher-level operations on the specified data types. Data abstraction should be included in the VHSIC HDL because it supports a design process style known as stepwise refinement. For example, a "stack" can be defined as a special data type with two operations, "push" and "pop", that the user sees. Any lower-level implementation details (e.g., that the stack is "really" implemented as a finite-sized FORTRAN array) is hidden from the user of the abstract data type stack. Such information hiding is valuable because of the limited availability of circuit designers to handle complex, low-level details.

3.2.2 Enumeration

The user should be able to enumerate the permissible values on symbolic data objects. This may be by enumeration as part of a set of values, as:

```
TYPE Pseudo-Boolean IS (one, zero, unknown).
```

3.2.3 Range

The user may also declare a range of values to be permitted, as:

```
TYPE Normal-Integer RANGE 0..65535.
```

Limits checking on the result of expressions can be used to enforce the bounds of data representation in assignment operations.

3.2.4 Precision

The user should be able to declare the precision of a real (floating point or fixed point) variable. The precision of a floating point variable is in terms of the number of decimal digits represented. As an example:

```
TYPE Normal-Floating-Point DIGITS 10.
```

The precision of a fixed point variable is in terms of a delta (resolution) value. As an example:

```
TYPE fixed-point IS DELTA-1/1024 RANGE (-2.0..2.0).
```

This would yield a fixed point number that is twelve bits long (sign plus one bit of integer value plus ten bits of fractional value).

3.2.5 User-defined Data Types

The user should be able to describe collections of data items grouped in several ways. The simplest is of homogenous nature as a one- or two-dimensional array, as:

```
TYPE Special Memory IS ARRAY (0...128, 0...32) of Boolean;
```

other associations may be quite useful where more than one type is grouped for the non-homogenous case, as:

```
TYPE System-Buss IS RECORD data, control END RECORD
TYPE data IS array (0...15) of INTEGER
TYPE control IS (READ, WRITE).
```

Character and bit strings may be built up as an array of enumerated data objects, as:

```
TYPE String1 IS ARRAY (1..80) OF Boolean
TYPE String2 IS ARRAY (1..132) OF Character
TYPE Character IS RANGE ('A'..'Z').
```

Arrays may be dimensioned to as many dimensions as the user desires.

The built-in VHDL data types should be held to an appropriate number of primitive types. These data types should be extensible by the user in a well-controlled manner. The built-in data types should include:

- Enumeration types--useful for Boolean and logic status
- Integer types
- Real type.

As with Ada, not only should precision be user-definable, but the underlying structure of a data type should be definable in detail so that an exact representation in the hardware can be made. Note that, unlike Ada, this exact underlying representation, if simulated, only needs to occur at the user-visible interfaces. That is, more efficient representation can be used within a simulation as long as these alternative representations do not affect the results of any operation.

In addition to the user being able to specify the exact representation of primitive data types, the user must be able to define new data types in terms of a collection (records as in Ada) of these primitive types.

Although generalized software pointers may be useful for describing software algorithms at high levels, only explicit hardware addresses are meaningful at the hardware level. These hardware addresses should be subtypes of integers rather than being of Ada-type "access".

However, given the complexity of VHSIC subsystems and the possibility of embedding traditional software concepts in hardware and/or firmware, the full generality of pointers should be included in VHDL to provide for an abstract description.

3.2.6 Deferred Definition

In order to serve as a design tool for a top-down methodology, the HDL should allow the designer to deal with elements of the design at a very abstract level at the high levels of system description. He should be able to "defer" decisions on implementation to later design stages. One style of deferral is to allow the specific number of components, of signals, of array elements, etc., to be represented symbolically, rather than by literal numbers. This is facilitated by allowing the designer to define a symbolic constant--an identifier declared to have a specific constant value. The identifier is used in the behavior or structural description for such things as array bounds. While the value of the identifier must be assigned specifically to be able to use the HDL description, it may be easily changed in one place, rather than in multiple places if a numeric literal value has been used.

Another aspect of deferred definition of objects is analogous to the progressive decomposition of design entities, but applied to signals interconnecting components. Signals should be allowed to have all the data types that behavior variables can have, including user-defined types. For example, this allows the designer to designate a signal as being an integer without specifying its explicit encoding in terms of a specific binary representation.

Note that if different components have different levels of abstraction applied to the signals joining them, then in order to run a simulation, in fact in order to check the interconnections, a transformation is required to connect the two types of signals. The transformation may be generated automatically by processing software, or may have to be generated by the designer. If the transformation is allowed to be generated automatically, the language definition should provide specific rules for this process. Coercion may be used to meet this need.

3.2.7 Strong Type Checking

One capability important for verification of design consistency is a compiler check for compatibility between declarations of variables and their use in expressions and function parameters.

Type checking ensures that the user is consistent in his interpretation of the various signals in the design.

The HDL should have strong type checking between operators, operands, and the corresponding results. Strong type checking allows "compilers" and simulators to ensure that operations on variables are compatible with the intended properties of the result types.

Type checking override facilities should be provided. Type checking may be done at run time and/or compile time.

3.2.8 Coercion

Coercion is a function which resolves type conflicts resulting from operations on differing data types. This includes converting one data type to another, as well as referencing one data type as an alternative data type. An example of an implicit coercion is the FORTRAN statement:

```
R1 = I * R2
```

where the integer variable I will be transformed into a real before multiplication with the real variable R2. In addition, at the hardware level it is important to be able to view variables as bit strings, as well as abstract definitions (integer, real, etc.) and vice versa. Thus arithmetic can be performed on a variable based on its abstract data type at one point in the behavior, and specific bits of the variable can be manipulated at another point in the behavior. The VHSIC HDL should provide a reasonable set of built-in coercions for the built-in data types as well as a mechanism for user-defined coercions between both built-in and user-defined data types. For the user to define a coercion, (1) a coercion function or procedure must be defined which, when the initial data type is passed to that function or procedure, the function or procedure will return the target data type, and (2) the compiler must be notified that this function is to be called implicitly when specified operations are applied to specified data types. It is hoped that coercion can be used to span multiple levels of design abstraction hierarchy, e.g., coercing from a group of Boolean variables or signals into an integer.

3.2.9 Reference to Attributes

An attribute is a predefined characteristic of a design entity and/or data object. For example, an attribute of a buss is its width.

In declaring data types the compiler maintains constants relating to the range of values expected, the subscripts permitted on arrays, and related properties of the data. These constants should be available to the Behavior section for constructing declarations of related data types. The mechanism as implemented in Ada is the Attribute capability as:

(memory'last)--relates to largest index into declared array.

3.2.10 Scope and Visibility

Scope is the parts of the VHDL description over which a declaration of a data object has effect. A named object is visible (can be legitimately referred to) if the reference is within the scope of the data object.

Scope for the I/O interface signals should be local to the design entity being described.

Scope for names of design entities and their terminals should be extendible to include components in libraries.

Name-qualification constructs should be included to resolve references to two data objects with the same name that are both visible.

The scope of a declared data object should be local to the design entity in which it is declared. This declared data object should not be known to any other design entity, even if another design entity is lexically within the definition of the design entity that declares the data object, unless it is explicitly defined as global in the design entity and enumerated within each of its component entities. With this "scoping" mechanism, data object visibility is defined by the contents of an argument list, and through explicit declarations and enumerations of global variables.

3.3 OPERATORS AND EXPRESSIONS

The HDL user should be able to manipulate the data objects by using a sequence of operators grouped for logical and order-of-evaluation reasons into expressions. The user should also be able to define additional operators and functions which will be treated by the system as its own set of primitive operators. Mechanisms should be provided to transfer control to routines that modify representation and format as needed for proper evaluation. Detection of fault or exception conditions should be allowed for user handling of unexpected arguments or invalid results.

3.3.1 Basic Operators

To adequately express behavior, the VHSIC HDL should have a rich set of operators.

The following operators are recommended:

logical operators - AND / OR / exclusive OR
relational operators - equal / not equal / less than / less than or equal / greater than / greater than or equal
adding operators - plus / minus / concatenation
unary operators - plus / minus / not
multiplying operators - multiply / divide / modulus / remainder
exponentiating operator - exponentiation.

3.3.2 Compare under MASK

MASK is an operator that allows the HDL to select important data bits or strings from "don't cares" or unimportant items. A hardware analog is the content-addressable memory which allows READ UNDER MASK and WRITE UNDER MASK operations. Search proceeds by examining the important areas of a string rather than by a pointing technique such as an address. MASK can be used, for example, in instruction sets to identify and use portions of each instruction, e.g., the operation code, the register field, the modifier field, etc.

This operator should be either user-defined or built into the VHSIC HDL.

3.3.3 User-Defined Operators

The user of the language should be able to represent functions that manipulate data items as operators. Capabilities should be provided to allow user-defined notations like:

$$C = A*B.$$

Arguments should be tested for stated type, required range of values, or elements of data sets as specified by the user. Its function should be transparent to the user whether an operator that he or she uses is built into the VHDL or is user-defined.

3.4 CONTROL FLOW

In writing procedural descriptions, the VHDL user will need to be able to define the selective and successive execution of procedures with respect to time. Repetition, selection, alternation, and directed branch options should be included as a minimum. Behavior descriptions may also need the capability to provide the user with the option to express parallel/concurrent execution of blocks of the code. Segments executing may need to exchange data or flags and also to prohibit other actions from occurring (mutual exclusion).

3.4.1 Control Constructs

A complete set of control flow statements should be included in the VHDL for algorithmic expressions. The IF, CASE, LOOP and EXIT, BLOCK, RETURN, and GO TO statements as defined in Ada include all necessary required statement types, and these types of expressions are well accepted in the industry.

3.4.2 Control Abstraction

Control abstraction is analogous to data abstraction (3.2.1) in its properties and usefulness, but it applies in the behavioral domain. For example, a machine could have a state known as "compute" that occurs after some preliminary initializations and before some final clean-ups and output. This compute state is a control abstraction for a number of substates, which might include multiple loops and subloops. The VHSIC HDL should support control abstraction.

3.5 TIMING

The VHDL shall provide the ability to describe the time at which output signals change value, in terms of delays after the times at which other "signals" change value. The "signals" may be either external signals, or purely internal to a design entity. (In a behavioral description of a design entity, an "internal signal" may be an artifice to aid in simplifying the behavior description. It may have no relation to a real signal, but it may strongly imply a particular implementation.)

The VHDL shall provide the capability to control at what time each statement in the description is executed. Sequences of statements will normally execute in zero time, but the capability will be provided to delay and resume execution at a later time, specified either as delay relative to the current time, or as a function of the time at which one or more signals change.

3.5.1 Synchronous and Asynchronous Time

The language should support synchronous, asynchronous and combined synchronous/asynchronous design.

An option would be to provide special explicit constructs for handling clock mode synchronous designs.

Examples of asynchronous events that are everyday occurrences include:

- interrupts
- input/output
- pipelining
- groupings of finite state machine
- peripheral interface busses.

Thus, one does not need to go to more advanced self-timed systems to find asynchronous events that are of crucial importance in common digital systems. It is essential that the VHDL describe parallelism and concurrency in a structured and relatively high-level manner.

3.5.2 Simulation Option Interfaces

The language should permit a variety of approaches to simulation:

- event driven
- clocked synchronous
- path tracing
- packet communication.

Each one of these simulation approaches has its advantages, depending on the desired

(a) simulation level

(b) simulation speed, and

(c) target hardware performance.

3.5.3 Timing Constraints

A method should be provided in the VHDL to specify timing constraints. Some of these timing constraints are illustrated below:

SET UP Time	The amount of time signal "A" must be in a stable state before signal "B" changes state.
HOLD Time	The amount of time signal "B" must be in a stable state before signal "A" may change state.
PULSE WIDTH	The minimum and/or maximum time a signal must remain in a stable state.

Timing constraints may be applied to input signals, output signals, or signals within a design entity. Timing constraints should be provided for at any level in the hierarchy. That is, timing constraints should be valid in the high-level behavior description as well as at the lower level of primitive component interconnects.

A facility should be provided at any level to make timing assertions to accommodate timing validation between levels in the hierarchy. For example, in the following high-level design entity the user may want to stipulate that the total delay between points A and B is no greater than 50 time units:

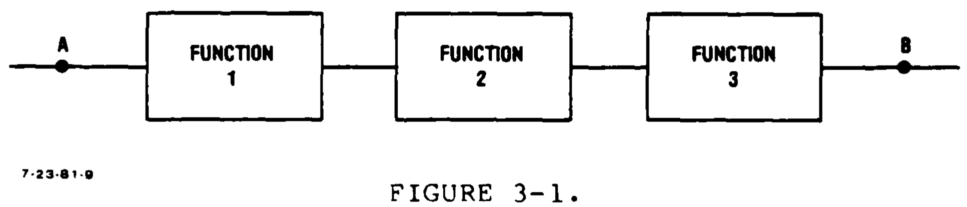


FIGURE 3-1.

where FUNCTION-1, FUNCTION-2, and FUNCTION-3 are high-level descriptions of operations performed by the design entity. As these functions are progressively more accurately defined at lower levels of the design hierarchy, it should be possible to compare the total delay to the original assertion made at the higher level.

3.5.4 Parallel Element Models

Synchronization should be supported in parallel operations. Synchronization may need to occur within a behavioral block or between behavioral blocks. Within a behavioral description the ability should be present to cause a given process to remain at its current state until a parallel process has indicated to the first process that it may continue. Synchronization between behavioral descriptions can be supported by passing parameters between descriptions.

3.5.5 Rise and Fall Times

Capability should be provided in VHDL for declaring and describing edge characteristics of signals. These should include but not be limited to rise and fall times, periodicity, leading and trailing durations, etc.

A method should be provided, at the logical interconnection of components, to associate timing information with each signal in the design entity. The user should have the ability to specify transition times between states in the simulation. For example, in the case of a four-state logic (HI, LOW, HIZ, UNK) there are 12 delays possible:

HI to LOW, LOW to HI, HI to HIZ, HIZ to LOW, etc.

In addition to specifying the normal delay between states, the user should be able to specify a range of delays. That is, three delays are allowed for each state change: NOMINAL, MINIMUM, and MAXIMUM.

3.5.6 Clock Definitions

It should be possible to declare the existence of one or more global clocks. If a single clock exists, the user should have the option of omitting reference to that clock in individual statements. It should also be possible to apply the language in contexts where explicit reference to a clock is required in individual statements.

3.5.7 Time Control Concepts

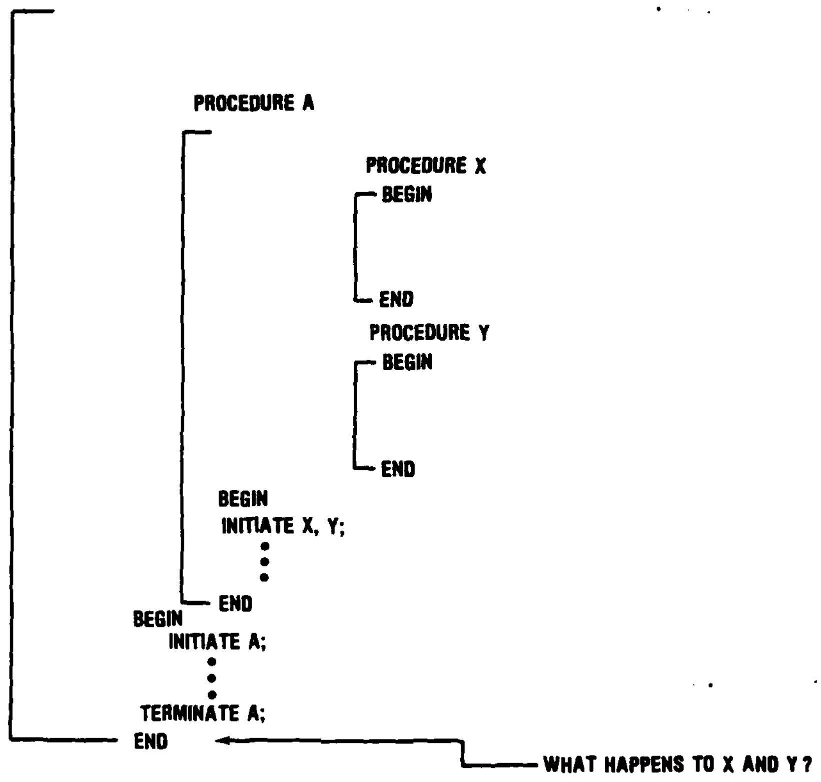
VHDL should be capable of unambiguous temporal descriptions. Provision should be made to allow formation of two types of systems with distinctly different time behaviors. One has data clocked through the system by means of a master clock or a locally-generated clock that controls a substructure. These systems are called synchronous in that data is gated through control points only at times determined by the clock. In the second type of system, data is permitted to propagate through the system at a rate limited only by the speed constraints of the logic structure itself. The second case is called asynchronous or self-timed in that clocks are not used to regulate the flow of data through the system.

Both systems can exhibit concurrency. In the clocked system, data may be clocked along two or more parallel paths during the same clock period. The problem arises when two or more parallel structures interact during the clock period. Path delays should be a part of the description and/or provision must be made to flag critical races. For the asynchronous system, the same signal may drive two or more parallel structures that interact, or two or more signals arrive at the same structure at the same time. In both cases, some method should be made to allow the user to specify which signal will proceed first.

3.5.8 Termination of Currently Executing Procedures

The VHDL should allow a procedure to have other procedures defined locally within its textual scope. These local procedures may be considered to be concurrently executing procedures.

Assume that Procedure A below has concurrent Procedures X and Y defined within its scope. X and Y are initiated and proceed to exchange data with A. At some point in time, Procedure A is terminated by its calling "parent". The user of the VHDL should define a semantic interpretation which states what happens to X and Y when their parent no longer is in existence. The dynamic activation of procedures implies that X and Y terminate when A is terminated.


PROCEDURE A
PROCEDURE X
BEGIN
END
PROCEDURE Y
BEGIN
END
BEGIN
INITIATE X, Y;
END
BEGIN
INITIATE A;
TERMINATE A;
END
WHAT HAPPENS TO X AND Y?

7-23-81-10

4.0 STRUCTURE

The structure of a design entity is its decomposition into interconnected components. The structure section should include declarations of the component types, instantiation of particular components, and listing of the interconnections. The VHDL should have constructs to handle iteration of component instances as well as interconnections. Generic instantiation of design entities should be provided to allow the user to specify default parameters.

4.1 CONNECTIONS

A structure description should consist of the specification of the components utilized in the design, the nets which interconnect their terminals, and the terminals of the design entity under consideration.

There are three different perspectives: (1) from the point of view of a terminal, connections refer to the net they are part of; (2) from the point of view of a net, connections are the list of terminals to be connected to form the net; and (3) from the point of view of the components, each of its terminals is connected to a net.

Each of these perspectives implies a different type of language construct. Their relative merits need to be evaluated and a decision made on which one(s) are to be incorporated.

4.2 GENERIC MODELS

4.2.1 Purpose

The purpose of a generic model is to make available, at any level of the structure description, multiple instances of a design entity, with the capability of variations upon a basic concept of that entity. For example, a generic model ADDER might be defined which makes available a family of adders with different numbers of bits in the arguments and different speeds of its interior components. The generic model is conceptually the hardware structure equivalent of the software subroutine or macro which uses arguments to modify its transfer function.

4.2.2 Parts

A generic model definition should include:

- the model name
- an ordered list of input and output signals, which form the functional connection with the rest of the structure
- an ordered list of parameters, which define the particular characteristics of a particular elaboration of the model
- declarations defining the variables used within the model definition
- one or more behavioral descriptions of the relationship between the inputs and outputs in its I/O list
- an optional structural description of the means by which the behavior is achieved in terms of interconnected instances of other design entities.

Sample syntax:

```
GENERIC [modelname] (I/O list, parameter list)
DECLARATIONS:
BEHAVIOR:
STRUCTURE:
ENDGENERIC modelname.
```

4.2.3 Communication

The functional communication between an instance of a generic model and the rest of the structure occurs only through its I/O list; all variables used within a model are local in scope and are independent of name assignments external to the model. However, for convenience and consistency, TYPEs defined outside the generic model may be allowable in defining these local variables.

4.2.4 Parameters

The parameter list may be defined as null if parameters are not required for a particular generic model. When a parameter list exists, the generic model definition should include for each parameter either:

a. a default value which is to be used if a particular application of the model gives a null value for the parameter, and optional limits on the value that may be assigned to the parameter, or

b. a specification that a null input value is not permitted.

4.2.5 Declaration Scope

The declarations associated with the generic model definition should have no effect outside the model definition.

4.2.6 Behavior

The behavioral description of a generic model should allow any feature and be governed by all constraints defined for behavioral descriptions elsewhere in this document. The behavioral expression will, in general, be a function of the parameter list.

4.2.7 Structure

The structural description of a generic module may contain all features defined for structural descriptions elsewhere in this document, with the following additions and restrictions:

1. Local block definitions of structure, and instances of other generic models may be used, and these may be embedded in conditional or iterative structures controlled by parameter

values or constants, but not by members of the I/O list.

2. Recursive reference to this generic model either directly or by reference to another generic model which refers to this generic module should be permitted only if:
   a. Adequate recursion control is provided in terms of parameter values or constants.
   b. Comments are included which provide a clear explanation of the physical interpretation in hardware of the results of the recursion.

4.2.8 Instance Names

The semantics of the language should provide for each instance of a generic model to be uniquely named. Such names are a function of an identification of the reference point, any applicable iteration indices, and the model name.

4.3 PERMUTABILITY

VHDL should allow the designer to describe "permutability" or "equivalence relationships" existing among the external connections of each design entity. One of the reasons is to allow the physical routing of interconnections to be optimized by taking advantage of equivalence relationships.

There are two types of equivalence: physical and logical. Two or more connections to a design entity are physically equivalent if they are connected to each other internally. Terminals are logically equivalent if they can be connected to any permutation of the signal net connections to them without affecting the output behavior of the design entity. An example is the inputs of a NAND gate, which are all logically equivalent. Logical equivalence can also exist between groups of connections, groups of groups, etc., to any depth of nesting.

4.4 GENERIC INSTANTIATION

A section of code should be able to be written with generic parameters. The values of these generic parameters can be deferred to compile time. At compile time this code is generically instantiated and expanded, based on the values given for the generic parameters. This is generally called "macro expansion". For example, a macro expansion is an instance of code that is a specific generic instantiation of a macro, or a variable is an instance of some type.

4.5 ITERATION IN STRUCTURE DESCRIPTION

VHDL will be of limited use if the designer must explicitly enumerate each occurrence of each component and signal net. VHDL should have powerful and easily used ways to express multiple instances of generic components and their connections, using constructs similar to iteration, conditional selection, and recursion as seen in programming languages.

As a minimum, regular, repeated arrays of components should be specifiable. Structural iteration may be implied using a subscript-like notation, where the subscript expresses a range of values, each of which corresponds to one instance of a generic component. For example, four instances of a generic component LATCH might be expressed by:

```
LBIT (0 to 3) : LATCH ...;  (TI HDL)
```

or:

```
LBIT : array (0..3) of LATCH (pseudo-Ada).
```

Iteration might also be expressed by an explicit iterative construct, like the Ada loop statement. An example of this style of syntax, in pseudo-Ada, is

```
for I in 0..3 loop
   LBIT(I) : LATCH ;
end loop.
```

Arrays of higher dimension than one are handled by multiple subscripts in the implicit method, and nested loops in the explicit method.

It may occur that an array of components is "almost regular", differing from "perfectly regular" only in that the first and last components differ from the middle components. It may be desirable to have an iterative construct that allows explicit definition of "first", "middle", and "last" components.

Conditional selection is analogous to "if-then-else" and "case" statements in programming languages. Predicates for conditional selection might be generic parameters or Boolean-valued functions on them.

While conditional selection by itself is not a powerful concept for replication, it enables structures to be defined recursively. Recursion is a technique that can easily generate multiple copies of a component. It is not clear whether recursion will be useful in many circumstances, but it is useful in the "N by N router problem" shown in Fig. 4-1. The problem

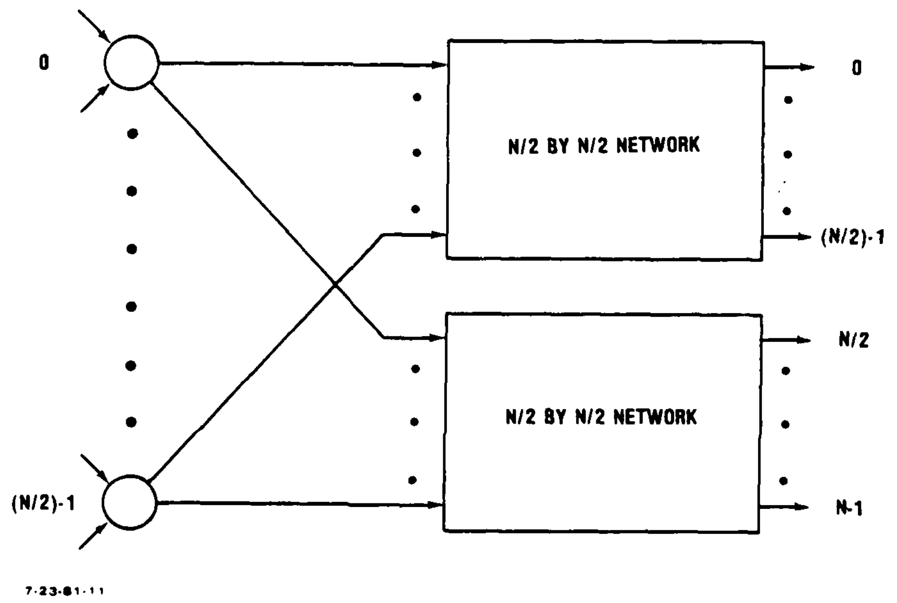


FIGURE 4-1. Recursive decomposition of an N by N network

is to decompose an N-input, N-output router into a structure of 2-input, 2-output routers, where N is a power of 2. The problem is solved by decomposing the N by N router into a structure of 2 by 2 routers and N/2 by N/2 routers. To correctly use recursion, the conditional selection construct is needed so that the recursion is not infinite. In addition, at least one generic parameter is required to pass the "current stage" of the recursion. The structural decomposition is shown graphically in Table 4-1. A description of the structure is given is HISDL, although a similar description should be possible in the VHSIC HDL.

TABLE 4-1.

```
%
% RECURSIVE DEFINITION OF AN N BY N ROUTING NETWORK
%
STRUCTURE ROUTING_NETWORK (N; IN INPUT[0:N-1], OUT OUTPUT[0:N-1])
  COMPONENTS COLUMN[0:N/2-1]: ROUTER
  %
  BEGIN
    IF (N = 2) THEN
      % SINGLE 2 BY 2 ROUTER
      COLUMN[0](INPUT[0], INPUT[1], OUTPUT[0], OUTPUT[1])
      ELSE
      % DECOMPOSE ROUTING NETWORK INTO A COLUMN OF
      % 2 BY 2 ROUTERS AND TWO HALF-SIZE SUB-NETWORKS
        COMPONENTS SUB_NETWORK[0:1]: ROUTING_NETWORK(N/2)
        FOP I = 0 TO N/2-1
          COLUMN[I](INPUT[2*I], INPUT[2*I + 1], SUB_NETWORK[0]. INPUT[I],
                     SUB_NETWORK[1]. INPUT[I]
          /OUTPUT[I], SUB_NETWORK[0]. OUTPUT[I]/
          OUTPUT[I + (N/2)], SUB_NETWORK[1]. OUTPUT[1]/
        ENDFOR
      ENDIF
    END
ENDSTRUCT
%
% 2 BY 2 ROUTER
%
CELL ROUTER (IN INPORT 0, INPORT 1, OUT OUTPORT 0, OUTPORT 1)
ENDCELL
```

9-2-81-2

Structural recursion need not be supported in the first release of the VHDL compiler.

4.6 PROCEDURAL MODELING

A procedural model of a structural description is a procedure that upon execution returns a structural description as specified by global variables and parameters passed to the procedure. Thus, it is a technique that could be used to implement the requirements of structural iteration and generic components.

4.7 EXTENSIBLE NUMBER OF PRIMITIVE COMPONENTS

The user should be permitted to define new primitives using VHDL syntax. This capability will minimize the need for actual language additions over time.

5.0 GUIDELINES

5.1 KERNEL PLUS EXTENSIONS EQUALS LANGUAGE

Figure 5-1 illustrates an implementation methodology for user-extensible language systems*. VHDL should include both a kernel of essential features and some mechanisms that allow the user to extend both the syntax and semantics of the language system in a controlled way. Extensions may be grouped together by a common purpose, much like the CALENDAR package of Ada defines a new primitive function CLOCK, and the abstract data types TIME and DURATION, including operations for their addition and subtraction. Such a grouping is represented by an arm of the starfish. Arms of the VHDL may well develop to support various application areas (e.g., signal processing), design styles (e.g., asynchronous or self-timed systems), or structured implementation techniques (e.g., gate arrays, PLAs, SLAs, etc.).

In those cases where a language feature is expected to be widely used (e.g., operations on the natural numbers) the kernel should be relatively rich and high level (e.g., including exponentiation). In cases where the proposed feature is relatively specialized (e.g., self-timed control primitives), the kernel should contain a limited number of low-level features. However, these features should be complete enough and flexible enough that the user can extend the language system without undue difficulty.

*(...which has been dubbed the Cape Cod Starfish in honor of the site of the IDA Summer Study on HDLs)

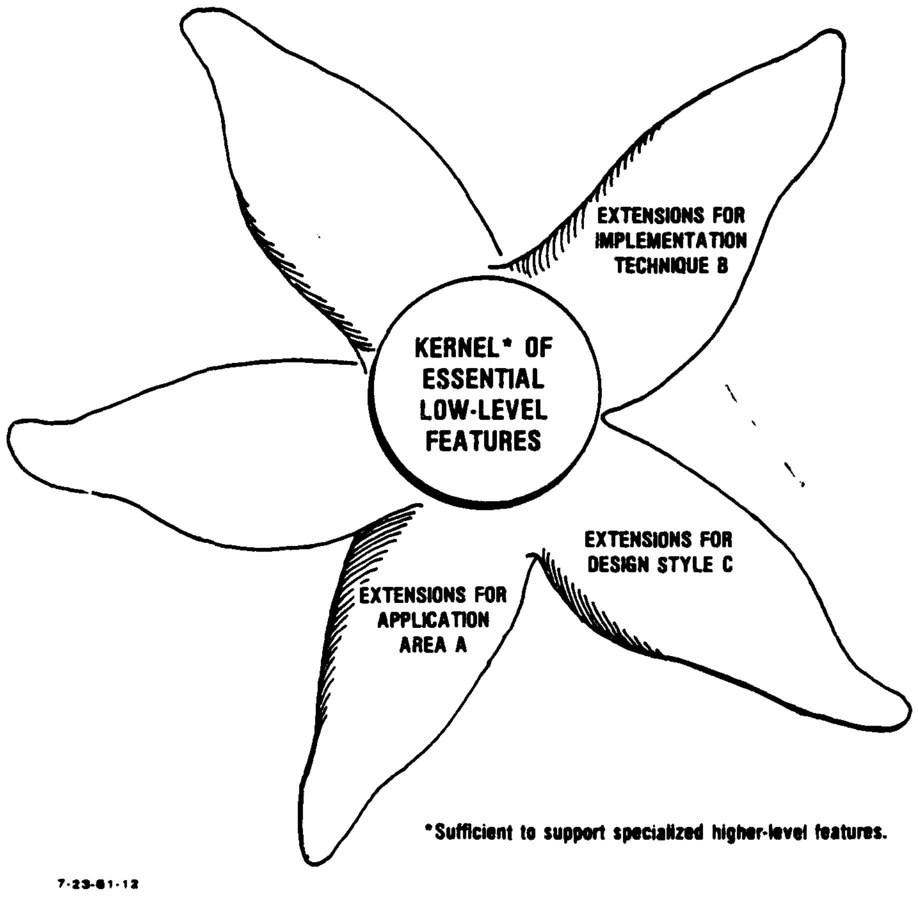

EXTENSIONS FOR IMPLEMENTATION TECHNIQUE B
KERNEL* OF ESSENTIAL LOW-LEVEL FEATURES
EXTENSIONS FOR DESIGN STYLE C
EXTENSIONS FOR APPLICATION AREA A
*Sufficient to support specialized higher-level features.
7-23-81-12

FIGURE 5-1. Starfish philosophy of extensible language systems

5.2 HIGH-LEVEL PARALLELISM

Descriptions of VHSIC chips may be done at a high level of abstraction. This high level may include the behavioral description of concurrently executing components which exchange data and control signals with each other.

For ease of description of these type designs, VHDL should provide a built-in mechanism for handling components operating in parallel and cooperating with each other.

Such mechanisms allow the description of behavior at a level well above that of physical hardware. Therefore, means must exist to verify that high-level mechanisms are correctly implemented by the lower-level structural descriptions.

5.3 VERIFIABILITY

VHDL should be verifiable. Any VHDL code should be executable so that someone other than the original author can verify that all information necessary for the complete implementation of the design is in fact contained in the code. By executable is meant the ability of the code to drive a suitable analytical tool. Such a feature is essential if someone other than the original designer is to be able to fully understand every single aspect of a given design. Too often the hardware description appears to be incomplete, is accepted by a second party and, months or years later, is found to be lacking some crucial piece of information without which the design cannot be completed.

Note that simulation is not the only method for achieving verification. A wide variety of analytical tools already exist for accomplishing the same objective (e.g., Boolean comparison, timing verification, etc.) and more are under development.

5.4 ONE LANGUAGE

VHDL should be a single language used to implement behavioral and structural descriptions at all levels. Inserts written in other languages (e.g., FORTRAN, Pascal, COBOL) shall not be permitted. In the determination of the details of the language, the following principles should be observed:

a. Economy in the number of statement types should be emphasized; a short list of powerful statements is preferred to a long list of diffuse and specialized statement types. When so used, all VHDL descriptions will be self-contained, given the standard list of primitives within the HDL.

b. Friendliness to the user should, in general, take precedence over compiler economics within the limitations of d.

c. Extensibility of the language should be achieved within the capabilities of the generic data statement, generic component, and user-defined functions and operators.

d. Terseness should be secondary to clarity and visibiity.

e. Statement syntax, where it derives from Ada, should not be "similar" to Ada, but should be adopted exactly as defined in Ada.

5.5 TECHNOLOGY INDEPENDENCE

Insofar as possible, features of the language should not imply any particular implementation technology in the hardware being described.

5.6 MEANINGFUL KEY WORDS

Key words and operators should be reasonably meaningful, and should not present new meanings to words or symbols commonly understood differently in other related environments.

5.7 COMMENTS

A comment starts with a unique delimiter and is terminated by the end of the line. It may only appear following a lexical unit or at the beginning or end of a program unit. Comments have no effect on the meaning of a program; their sole purpose is the enlightenment of the human reader.

5.8 COMPILER DIRECTIVES

A standard construct should be provided for conveying instructions to software that processes the VHDL code. Such instructions may be either implementation-defined or language-defined. Language-defined instructions are those for which the language specification prescribes a meaning, and all processing software must interpret that meaning as specified. Implementations may allow additional instructions as desired. Processing software should be able to interpret or ignore implementation-defined instructions without changing the meaning of the VHDL code.

5.9 EXPRESSION OF INVARIANCE

In the behavior description of VHDL, provision should be made to construct expressions that are invariant, i.e., are always true. The invariant expression may be global or local to some part of the behavior description.

One class of invariant expression is that which is user-defined as an expression of the particular hardware system under design. Another class may be timing constraints between design entities imposed by the system requirements.

5.10 LR(1) GRAMMAR

The VHDL specification shall use a formal grammar to specify the language syntax. The specification should use BNF notation, or a close analog. The grammar should conform to the class of grammars called LR(1). The reason for this requirement is that it allows the employment of existing parser-generator ("compiler-compiler") tools to generate software to process VHDL code.

Note that the requirement does make it more difficult to specify the grammar and does impose some constraints on the language. These constraints are probably not objectionable.

5.11 EXECUTION CONFLICTS

VHDL descriptions should have specific unambiguous semantics. Descriptions that are self-contradictory or do not allow for an unambiguous interpretation are in error. Wherever possible, descriptions in error should be identified by processing software. It is not acceptable to have processing software take simulator-dependent or technology-dependent actions in any circumstances that are not specifically called out by the language specification. To do otherwise conflicts with the important goal of portability.

The semantics even of an unambiguous HDL are open to technology-dependent ambiguities. For example, assume in the the example below that A is a bus and X is a value imparted to the local environment:

```
A:=1 AFTER 2 nsec;
A:=0 AFTER X+1 nsec.
```

If X has the value 1, then a technology-dependent ambiguity arises. If the system is implemented in TTL, then the above condition may cause the bus to overload and burn out. However, in MOS technology the bus may simply become undefined.

Such conditions may not be detectable at compile time (any more than array overindexing is detectable at compile time).

The VHDL user shall have the responsibiity for indicating, via a compiler directive, the semantic interpretation or exception-handling procedure is to be invoked when this occurs.

5.12 MULTI-VALUED LOGIC

The language should not restrict the number of logic states. The user of the language should be able to enumerate the number of logic states at any level of abstraction. Multiple logic states implies multi-valued arithmetic. Care should be taken in defining the meaning of operator and coercion so as not to exclude multi-valued arithmetic.

5.13 INCREMENTAL COMPILATION

It is recognized that the implementation techniques of incremental compilation (statement by statement) and piecewise compilation (section by section) provide the VHSIC designer with a valuable degree of interactiveness. Therefore, nothing in VHDL should preclude or place undue hardship on such implementations.

5.14 EXCEPTION HANDLING

An exception is a run-time occurrence (e.g., type mismatch, divide by zero) that does not allow processing to proceed in a "normal" manner. From the point of view of the user, exceptions should be handled automatically and/or error messages should be meaningful. From the point of view of a program, a mechanism should be provided to trap exceptions (i.e., to transfer control to a specified point when an exception occurs).

APPENDIX A

WHDL

WHDL

This report has been written to describe some proposed changes to the TI-HDL to conform to the recommendations of the IDA Summer Study which collectively specify a new language, VHDL. Since some, but not all, of the VHDL features are incorporated it will be referred to as WHDL*. The reasons behind this (omitting some VHDL features) were in trying to preserve the conceptual integrity of the TI language as it now stands. Many of the Summer Study suggestions would have torn this conceptual fabric. The subcommittee felt that WHDL should be able to serve the immediate (Phase II) needs of the VHSIC program. This document is not meant to design a new language; rather its intent is to point out where the TI-HDL falls short of the requirements document and suggests some changes that would try and bring the TI-HDL up to the current requirements.

The section numbering of this report, starting at Section 3.0, corresponds to the numbering of the TI-HDL manual. Comparisons between the manual and report should be easier. No final syntax is implied in this report.

The members of the committee which prepared this section were:

David Ackley	Scott Perkins
Manuel d'Abreu	Richard Sanders
Ernest Codier	Myke Smith
Clement Leung	Bill Stewart.

*For Woods Hole HDL.

1.0 VHSIC-HDL BLOCK STRUCTURE

1.1 The VHSIC-HDL is a block-structured language which permits the description of hierarchical designs. An outline of the VHSIC-HDL blocks is given in Figure 1. There are several types of blocks, namely:

DESIGN blocks describe the topmost block in the design hierarchy, this is the root of the design tree. A DESIGN block may only be instantiated once.

MODULE blocks describe design entities which are blocks at other than the top level. They can be multiply instantiated.

These blocks (DESIGN, MODULE) describe designs with a variable behavior and/or structure which are fixed by elaboration at the "time" they are instantiated. These blocks may also exhibit a fixed behavior and/or structure.

PROCEDURES which are used to enhance the readability and coding ease of behaviors and are analogous to procedures in PASCAL or subroutines in FORTRAN.

FUNCTIONS which are commonly used logical or arithmetic entities which return only a single type. All function arguments are input values that are "evaluated" upon function invocation. Since global variables are outlawed, functions will have no side effects. There are several "built-in" functions in WHDL.

1.2 DESIGN and MODULE blocks consist of seven sections of which all but two are optional. These are:

GLOBAL DECLARATIONS (optional)
ENVIRONMENT (optional)
BEHAVIOR
STRUCTURAL
TIMING (optional)
TEST (optional)
PERSONALIZATION (optional)

Either BEHAVIOR or STRUCTURE must be provided.

1.3 The structure herein defined is intended as a description wherein each block may be compiled separately. Global declarations, see 1.6, may appear in each block by virtue of an INSERT (or equivalent) verb which includes the globals defined in the DESIGN with the MODULE block. Binding of these structures for simulation may be achieved via a loader. Globals defined in one DESIGN may not be referenced by another DESIGN even though they are loaded together. Thus, the DESIGN (and all its subordinates) forms a scope of access for all of its globals. Communications between designs are established exclusively through the I/O lists.

1.4 A DESIGN or MODULE block will include behavioral descriptions which, along with its referenced procedures and functions, are self-contained, i.e., they form a complete behavior for that level of design. As that block is decomposed, the behavior of each subordinate is prepared and the structural section of the block is coded. The structural description of the block and the behavioral descriptions of the subordinates together provide a description which is functionally equivalent to the behavioral description of the block.

An important point to note is that the sections in a DESIGN block override the sections in the MODULEs that make up a design. For instance, the ENVIRONMENT section for a MODULE must be equal to or some "subset" of the DESIGN block's ENVIRONMENT. Another example is using the DESIGN block's PERSONALIZATION section to change the contents of some memory that may have been previously initialized by a MODULE's PERSONALIZATION section. In this way a MODULE could represent a micro-coded element and the DESIGN block changes its micro-code.

1.5 Global declarations include user-defined data types, constants and universal names. The user-defined data types define data abstractions which characterize signals, constants and variables for that design. Global constants provide a means of deferring design decisions by encoding the behavior in terms of named constants which may be easily changed. Universal names define signal names, buss names, voltage names, etc. which are used throughout and therefore are common through all levels of the design. Their purpose is soley for documentation purposes and to insure that the same name is always used in argument lists. For example, Vcc is not called power in different routines or modules. The existence of a universal name in the global declarations does not negate the requirement for that universal name to appear in an I/O list. Procedure, function, module and macro names may also be global names depending upon how instantiation and elaboration syntax is developed.

There are no global variables. All items must go through a signal list with an attribute that says whether it is NODE, IN, OUT, INOUT. This is for clarity and debugging. See Legard 77 and Wulf 73.

1.6 Macros are elaborated to produce specialized behavioral or structural VHSIC-HDL code. This may be viewed as a mechanism for "tailoring" the user's code during a "pre-compilation" process.

1.7 The ENVIRONMENT Section is intended as a mechanism via which general design requirements and characteristics may be expressed. Such things as power supply voltages, power consumption, space, thruput, etc., may be expressed. Technology selection may be specified. For cases where design I/O signal levels are specified, attributes used in the signal I/O list may be herein defined. This section is intended for use within design blocks only. Checks should be made on the consistency of the ENVIRONMENTs for each block.

1.8 The BEHAVIOR Section contains a procedural (algorithmic) description of the behavior of the block. It consists of sub-sections containing (local) declarations and statements. The declarations define data types, constants, variables, registers, terminals, clocks and external procedure or function names which are peculiar to that behavior. This is followed by a series of sequential and/or concurrent statements. A wide variety of such statements is provided in order to satisfy descriptive needs ranging from tne design concept to a behavior which is directly mappable into hardware.

1.9 The STRUCTURE Section is an expression of the design block in terms of lower level modules.

1.10 The TIME Section contains assertions relating to the timing relationships of the input and output signals. As a means of increasing the utility of this section, we suggest the inclusion of labeled statements. These statements are either expressions or triplets of numbers. These labels are used in the BEHAVIOR section to point to an expression that is to be evaluated for the incremental time calculations. Examples are (MIN, NORMAL, MAX), a distribution function, etc.

An abstract I/O timing model for a block can be defined.

1.11 The TEST Section contains stimuli and results required for functional and perhaps fault tests. The form and existence of these statements are negotiable (between contractors). Functional tests may also include assertions upon the timing relationships of stimuli and results.

1.12 A PERSONALIZATION section is required to describe the contents of RAM, ROM, PLA, etc. It is conceivable that a designer may wish to describe microcode behaviorally as existing in a composite memory. It will be necessary, therefore, to express the partitioning of that code into parts in the physical structure. A specific syntax is not suggested.

1.13 It is important that a data abstraction capability (ala Ada) be provided. This capability allows the user to define his own data types and the (allowable) operations upon those types. With this capability it is possible to relegate LOGIC and VALUE (TI-HDL data types) constants to user-defined types. A mechanism for defining the operations upon user-defined types is suggested in Ada. It should be noted that not all the features of software-oriented data abstraction are necessary for an HDL. For instance, finalization routines and types as parameters.

2.0 SUGGESTED SYNTAX CHANGES

2.1 Instantiation

In order to make the code more readable at the block level we would suggest the following (approximate) syntax:

```
[DESIGN BLOCK] name (signal_list)
NODE (node-signal-list)
IN (in_signal_list);
OUT (out_signal_list);
INOUT (inout_signal_list);
```

The signals in signal_list are in the same order as signals in the STRUCTURE section. The signal names in signal_list are the same names used in the NODE/IN/OUT/INOUT_signal_list. The signals in the NODE, IN, OUT, INOUT lists can have a general attribute list as discussed in Section 2.3. The entire NODE,IN,OUT,INOUT list can have an attribute list which is common to all the listed signals. (NODE implies no directions for signals, this should only be used for circuit-level descriptions and lower).

2.2 Generic Modules

Elaboration is a technique by which code can be conditionally tailored based on a set of parameters. A common implementation of this technique is called "conditional macro expansion." For our purposes a fairly powerful syntax, beyond simple text substitution, is required. Two examples are the IBM MACRO Assembly Language and DEC's MACRO Assembler. A token type of processor appears to be of sufficient power rather than a more powerful character string processor.

There should be two types of elaborations, one which generates in-line (behavioral or structural) code (see 1.6) and one which generates entire blocks for subsequent compilation. It is conceivable that both aspects could be included within the same elaboration, perhaps using a LOCAL and REMOTE attribute on the code blocks. The syntax should be identical for both types. An example of a block that is elaborated for subsequent compilation is the MODULE block.

```
MODULE  module_name (signal_list; elaboration_parameters)
```

An example of inline code could be:

```
     MACRO shifter (A,B,C)
           .
           .
           .
     ENDMACRO
```

Where A, B, and C are the elaboration parameters.

2.3 Literals should be any-base and could use the following syntax: 16#DF9E#; hex, 2#101#; binary, 8#7034#; octal, 2E6 implied decimal integer, 3.14159 implied decimal real, .271828E1 etc.

In addition, a general syntax for user-defined data types should be developed. Assuming user-defined types 'point' and 'value', their literals may be

```
(X,Y):point
(15.7, UF):value
```

2.4 Attribute lists should be general and free-form in the nature of TI's TEXT attribute, for all attribute lists, rather than be particularized for each use. It would be the job of the application programs to scan an attribute list at any particular level to determine if any pertinent attributes are defined.

```
@ (attribute-list)
@ (TTL, ICSB=NOM_MIN_MAX,FANOUT=6)
```

NODE, IN, OUT and INOUT should not be elements in an attribute list. They should be pulled out to clarify exactly what the signal is to do, rather than being tucked away in a long list of attributes.

BEHAVIOR levels should be able to have an attribute list. This list could, for example, specify the level of simulation for this specific block.

2.5 Implicit data types are:

```
INTEGER (MININT..MAXINT)
REAL (MINREAL..MAXREAL)
DOUBLE PRECISION (MINDREAL..MAXDREAL)
BOOLEAN
CHARACTER
RECORD
ARRAY
```

- A mechanism may also be needed to allow variant forms of records.
- Coercions between different implicit data types need to be defined.
- Coercions between different user defined data types are probably best handled in the data abstraction mechanism.
- Character is not meant for I/O rather it is a mechanism for extending binary representation.

2.6 Parameters

Parameters in the TI-HDL are a means of defining a value and an associated unit of measure. There will need to be a way of representing this in a natural fashion. Using a RECORD (like PASCAL) we would define a capacitor having a 10pF value as (10,pF). This is not very pleasing. We need to be able to write and parse (10pF).

3.0 VHSIC-HDL SECTIONS

3.1 STRUCTURE Section

A component should be one of the following forms:

component_name: module_name (signal_list)
where each signal is explicitly defined and typed, with its own attribute list, as defined in Section 2.3.

OR

component_name: module_name (signal_list; elaboration_parameter_list)
Elaboration_parameter_list is used to elaborate this specific instance of generic-name. Signal_list has the same ordering as the signal-list in the block definition. The specific NODE, IN, OUT, INOUT signal_list with attributes should appear on any listing for visual verification and future reference. This can de derived from the instantiated block by a cross reference lister.

The component_name need not be supplied. A translator generated name will be supplied.

The iterative structuring capabilities needs improving to explicate subscript iteration. The "first", "middle", and "last" elements in an iteration should be able to be handled separately.

3.2 BEHAVIOR Section

The following section describes suggested changes to the current TI-HDL.

3.2.1 Standard Models

The standard models should not be explicitly stated in the language; this is more appropriately a "library" and binding problem.

3.2.1.1 Program Header

"FUNCTION" and "LOGICAL" should be deleted because they indicate a specific type of simulation.

3.2.1.2 Behavior Declarations

```
VARIABLE var_name OF TYPE_name  :  initial_value
CONSTANTS const_name OF TYPE_name  : = structured_constant
REGISTER reg_name of TYPE_name  :  initial_value
TERMINAL term_name of TYPE_name
MEMORY mem_name of TYPE_name  :  initial_values
CLOCK clk_name [duty-cycle, number phases]
TYPE type_name=type definition
EXTERNAL externally_defined_procedure_name (args)  :  type_name
LABEL identifier
```

-REGISTER defines what are registers for register transfer simulation. Objects of type REGISTER can only be assigned with the transfer operator.

-TERMINAL defines what are terminals for register transfer simulation.

-MEMORY is for purposes of memory management at simulation time, and it aids the hardware synthesis application program.

-CONSTANT allows defining structured constants.

-TYPE is used to define new types. All types must be defined. No anonymous types. A type can be enumerated, a ordered set of values. (Problems with enumerated types and solutions in Moffat 81 and Enum 81). Types should be parameterized, but types as parameters are an open question.

-CLOCK will be used for defining specific clocks.

- LABEL defines identifiers that are used as labels in both the BEHAVIOR (GO TO and DO) and TIME sections.

-EXTERNAL defines externally-compiled procedures or functions. This is not a declaration.

-We propose that data abstraction capabilities similar to Ada's package be implemented.

-Use of predefined characteristics of a data object, such as its size, by using symbolic names. This is the Ada "attribute."

3.2.1.3 Program Statements

-Propose that a symbol be used to denote sequentially ";" and a symbol, " | " upright bar, or exclamation mark, "!", be used to denote concurrency. This is much clearer than a comma. (The upright bar or the exclamation mark will be chosen depending on the type of terminal used.)

-Case Statements

- o Case Label: The "by integer" option is valid for integers only.
- o The case "expression" and "case-label" should be expanded to handle integer, boolean, enumerated types, etc.
- o Case Labels should allow enumeration of elements in the label, for example:

  (5,8,9,10,11): statement list

Guarded Commands

The notion of guarded commands (Hoare 78) may be used to extend the TI/HDL in describing how a behavior block synchronizes with input signal changes and foresees new input values. A guard is a boolean condition. Some examples of guard conditions are:

-a ready line (used to implement a hand-shake protocol) undergoes a 0 ---> 1 (or 1 --->0) transition

-a clock signal undergoes a 1 ---> 0 or 0 ---> 1 transition.

-a state variable has a certain value

-AND/OR of guard conditions

A behavior block expressed using guarded commands is organized as follows:

```
BLOCK...
<declaration of variables and their initialization>
[[guard_1]]:  BEGIN  <body_1> END;
[[guard_n]]:  BEGIN  <body_n> END;
END block;
```

When this behavior block is first entered, its variable are instantiated and initialized. Thereafter, it repeatedly evaluates guards, picks one that is satisfied, and executes the corresponding body.

1) Mutual Exclusion

If more than one guard is satisfied due to a new input signal change or a new state variable update, one of them is selected by arbitrary choice and its associated body executed.

2) Except for issues of fair arbitration (every active guard will have its associated body executed at some time, and cannot be locked out forever), the guard commands are equivalent to:

```
IF [[guard$_1$]] THEN BEGIN  <body$_1$> END
ELSEIF [[guard$_2$]] THEN BEGIN  <body$_2$> END
ELSEIF [[guard$_n$]] THEN BEGIN  <body$_n$> END
```

3) Self-Timed Implementation of Packet Systems (Petri Nets too)

Example

```
BLOCK...
<declaration and initialization>
[[arrived (packet-port-1)  ...  arrived (packet-port-n)]]
     BEGIN <process> END;
[[acknowledged (output-port-1)  ...  acknowledged (output-
     port-m)]] BEGIN <send output> END;
END block;
```

4) Finite State Machines

Example

```
BLOCK
   s:integer   :=0;
[[s=0]] BEGIN initial state END;
[[s=1   <some input condition> ]]
   BEGIN .....S:=a; END;
   .
   .
   .
[[     ]] ...
END
```

5) Clocking

Example

```
BLOCK
<declaration and initialization>
[[clock-phase 1   state=0]] BEGIN...END;
[[down transition (phase 2-clock)   state=extend-cycle ]]
   BEGIN...END;
   .
   .
   .
END
```

-Incremental Time

Sequential statements can have delays assigned to their execution. Delay times can be accumulated to depict the total delay due to the execution of a block of sequential statements. The accumulated delay can be used to schedule value change to output signals. The following illustrates the use of the incremental time.

```
BLOCK ALU
     IN:  (A,B,C,D,CONTROL),
     OUT:  (E);
     IF (CONTROL) THEN BEGIN
                         Z=A+B <5>;
                         Y=Z.AND.C <2>;
                         E=.NOT.Y <1>;
                         END
                    ELSE BEGIN
                         Z=A-B <3>;
                         Y=Z.XOR.C <2>;
                         X=Y+D <5>;
                         END=.NOT.X <1>;
                         END
     END(ALU);
```

The values enclosed in < > denote the delay due to the execution of the statement. If a label is found within < >, then the labeled expression is in the TIME Section. This expression can be an arithmetic function or a set of values.

-WAIT statement

This statement will interrupt execution of a particular "block" and schedule its resumption after the specified time. When the time specified by the WAIT statement is elapsed, the "block" will resume execution at the statement where its execution was interrupted.

When a "block" is in WAIT, any input changes will not resume its execution; however, when the "block" comes out of WAIT, execution will resume with the new input changes, if any.

The following example illustrates the use of the WAIT statement.

```
    BLOCK JUNK
    IN:  (A,B,C,D,E,);
    OUT:  (SUM,COUT);
    IF (CONTROL) THEN BEGIN
                      Z = A+B+C <5>;
                      WAIT <3>;
                      SUM=Z.OR.E<10>;
                      END
                 ELSE BEGIN
                      Z=A-B <2>;
                      WAIT <4>;
                      COUT=Z.AND.E <10>;
                      END
END;
```

The WAIT statement in the "THEN" clause will halt execution for three time units. The internal variable Z will be scheduled before the WAIT is executed. After the WAIT time elapses, execution will resume with the statement following the WAIT.

-SCHEDULE statement

We propose that this be replaced by the OUTPUT statement. The semantics of OUTPUT will be defined. The OUTPUT statement does not imply the existence of a simulator like the SCHEDULE statement does.

The OUTPUT statement specifies a future change of value for an output signal. The following illustrates its use:

```
SUM = A+B <5>
OUTPUT SUM AT <8>
CARRY = A.XOR.B <3>
OUTPUT CARRY
```

In the first instance of OUTPUT, SUM will have its new value at (5+8) time units. In the second instance of OUTPUT will make CARRY get its new value at (5+8+3) time units in the future. Note that in the case of sequential statements the delay is accumulated. The default delay for OUTPUT is one time unit. If the delay for the OUTPUT statement is a label, then the delay will be retrieved from the TIME Section. The delay for the OUTPUT statement can also be an arithmetic function.

-Special statements

The special statements should be deleted. TESTPATTERN in the TI-HDL can be done in the VHSIC-HDL TEST Section. BREAKPOINT, the TI-HDL constructs should be done at simulation time and not be a part of a BEHAVIOR section.

3.2.1.4 Expressions

-VCHANGE is a boolean function. We find it necessary to have a RISE and FALL function to indicate whether a signal is rising or falling.

-A transfer operator (<---) is required for assigning data to objects of type REGISTER. The right hand side of the transfer expression should be able to include a condition(s) IF-THEN-ELSE or a CASE.

-Define a reduction operator that works in conjunction with the logical operators (AND,OR,XOR,EQU,NAND,NOR) and that work over a boolean vector, e.g., AND/vector.

-Operators should include exponentiation. The NOT operator in Table 8 is not a binary operator.

-Operator Precedence - expand this to include reduction and exponentiation operators. Exponentiation should have precedence between that of unary operators and multiplication. Reduction should have the same precedence as a unary operator.

-Arithmetic functions such as the trigonometric and logarithmic functions should be included.

-For enumerated types and characters, several component selectors are required, for instance, finding the value succeeding X in the enumeration or character set. There are several different approaches, that of PASCAL (PASCAL 75), Ada (Ada 80) and Euclid (EUCLID 77).

-Delete the PARITY operator since the reduction XOR accomplishes this.

-Add a COMPARE UNDER MASK function that would take a mask of 0,1, and ? (don't care) and compare it to a boolean vector; if the comparison is correct return a true.

3.2.1.5 Concurrency

The TI approach to describing concurrency is to push it out of the BEHAVIOR into the STRUCTURE. The mechanisms for describing concurrency in the BEHAVIOR section are therefore not very powerful. The concurrency capability could be expanded to permit the description of very complex control threads in the BEHAVIOR section, mechanisms for this are described in the addendum. This, we believe, would add to the complexity and burden of the simulator. (It should be strongly noted that structural decomposition does not necessarily mean a physical decomposition but that it can as easily be used to describe logical decomposition.)

Sequence concurrency in the TI-HDL is accomplished by executing blocks in parallel. The execution of sequence is done in a 'step-lock' mode (TI terminology). The following example, demonstrating sequence concurrency with the DO statement, comes from the TI manual.

```
E.G.,         'DI';
               DO A, DO B, DO C;
              'D2';
              'D3';

          A:  BEGIN
              'A1'; 'A2'; 'A3'; 'A4';
              END
```

```
    B:  BEGIN
        'B1';
        END;
    C:  BEGIN
        'C1'; 'C2'; 'C3';
        END;
```

The statement execution sequence is:

STEP	STATEMENTS IN PARALLEL
1	D1
2	A1, B1, C1
3	A2, C2
4	A3, C3
5	A4
6	D2
7	D3

The execution of statements in a sequence (D1, D2, D3) is suspended when a DO is encountered until the sequence within the block is complete. In concurrent blocks, the longest sequence must complete before continuing the original sequence. If within a block there exists another BEGIN-END pair this is treated as a single statement in terms of sequence concurrency. For instance, if in the above example statement 'A1' is a BEGIN-END pair then statements 'B1' and 'C1' would be executed in step-lock with the first sequential statements of 'A1' and then the rest of 'A1' would be executed before going on to step 3.

A question was raised as to what happens if the following example is executed:

E.G., DO A, DO B;

```
A:  BEGIN                      B:  BEGIN
      .                              .
      .
                                   DO C;
                                     .
    DO C;                            .
      .                              .

    END                            END;
                   C:  BEGIN
                         .
                         .
                         .

                       END;
```

Since A and B can be of arbitrary complexity in terms of control statements, one cannot predict without a great deal of analysis, whether C will be invoked simultaneously by both A and B or whether C will be invoked while currently executing A or B. TI says that the statements are lined up, for the step-lock mode execution, dynamically. In other words the "lining" up of the statements is not done in a static fashion at the time the source code is translated rather it is done dynamically by the simulator at execution time. The user must be aware that if the set of sinks and the set of sources for assignment statements, executed in sequence concurrency, is not disjoint s/he is inviting problems when a physical implementation is realised. The simulation will always be deterministic, but if the realization is with asynchronous hardware, the hardware's response may not be deterministic.

To insure determinacy of the hardware in the above example, only one invocation of block C at any one time can be permitted. This is mutual exclusion. An optional attribute of CRITICAL for a block of code can be implemented. A CRITICAL attribute will inform the simulator that only one invocation of this block of code at a time is permitted and any other requests for the code are to be queued up until the block is idle. Using the attribute for mutual exclusion allows the use of the concept and does not require a description of a mutual exclusion implementation. This can be deferred until much later in the design process.

There is an unpublished and yet unimplemented feature of the TI-HDL for synchronizing sequence concurrency. The syntax is:

```
identifier1: SYNC identifier 2, identifier 3, ...
```

Identifier 1 is optional on all but one of the SYNC statements. The identifiers are not labels, in other words the identifiers cannot be objects of a 'goto'. The semantics of this statement are as follows. Any two blocks, executing in parallel, can be synchronized by saying

```
identifier: SYNC identifier
```

in each block. The identifiers in each of the two SYNC statements are the same. When a control thread reachs a SYNC statement, it waits until another SYNC statement is executed with the same identifier. Then both control threads proceed on executing. If multiple identifiers are listed as the object of the SYNC statement then if any one of the identified SYNC statements is also executed the two control threads proceed on executing. The list of identifiers is an 'or' condition. Thus, only two SYNC statements can be synchronized at a time. This ought to be modified to allow the synchronizing of an arbitrary number of control threads. This could be done by separating the identifiers with the 'or' or 'and' operators.

Step-lock mode execution cannot be carried out if incremental time is used. Using the first example, add an arbitrary time to the execution of the sequential statements. Now, map this into the statement execution sequence. Take step 2, for instance, the statements 'A1', 'B1', and 'C1' are to be executed in parallel. If we give arbitrary times 5, 3, and 7 as execution times to 'A1', 'B1', and 'C1', respectively, then what does this mean in a step-lock mode of execution? A new set of semantics needs to be developed to describe the simulation of statements which have an arbitrary execution time. An approach to handling this problem would be to disallow time within a block, which is the obejct of a DO or is to be executed concurrently with another block, or allow a time specification for the entire block. For instance, DO A <8>;

None of the statements in block A could have a time specification. But block A is specified as taking 8 time units.

More investigation is needed into the defining the proper constructs for concurrency. TI's sync is insufficient. The constructs should define concepts and not imply implementations.

3.5 Time Section

-Specify timing constraints that are to be checked for, e.g., set-up and hold time.

-Specify details of timing "variables" used in the BEHAVIOR section, e.g., (min, Nom, Max) or a distribution function.

-Abstract I/O timing model for a block. This model is used for resource analysis type simulation where only I/O timing is of interest and no I/O data transformations are required, e.g., constant Gaussian with mean and standard deviation.

3.5.1 Delays

-We find it necessary to be able to specify delays in terms of a nominal value, standard deviation and statistical distribution. The distribution can be specified as normal, Gaussian, etc.

-Delays should also be specified as a function of loading. Also, we find the necessity to be able to specify media delays (i.e., delays due to layout, etc.) as a lumped delay associated with an output signal.

Addendum

A method of handling behavioral concurrency

Currently, the TI-HDL cannot support the description of arbitrary concurrency in a behavioral description. Sequence concurrency can be initiated by DO A ! DO B; This has an implied forking of A and B and an implied join after the completion of A and B. The main control thread does not advance until after all "forked blocks" are terminated. This is a limited means of expressing control threads. We propose the following attributes for a block.

PROCESS - the block is to be executed as an asynchronous control environment. An activation of a PROCESS block starts the block executing concurrently with the caller.

CRITICAL - the block contains an artibration mechanism so that one and only one activation of the block can be in progress at the same time. Activations are queued if the block is already active.

PROCESS and CRITICAL are independent attributes. The former controls the continuation of the callers, the latter controls concurrent callers. When both attributes are present, CRITICAL takes precedence. That is, the caller of a CRITICAL PROCESS block is delayed until the block is free or idle before it continues executing in parallel. Attempting to activate an already active non-critical block is an error and it yields unpredictable results. These attributes do <u>not</u> imply any implementation but represent concepts in a clear and unambiguous manner.

The "DO" statement then operates as a fork operation on PROCESS blocks. These spawned control threads can be reunited to the main or spawning control thread by JOIN (Labeled_block1, Labeled_block2, ...). They can also be arbitrarily terminated by any other control thread by issuing TERMINATE (Labeled_process_block). The process_block can terminate

itself by reaching its "END" statement. The JOIN operator forces the main control thread to wait until all listed process blocks are through. The "END" without an associated JOIN implies the control thread is simply stopped, a dead end. It has done its duty but the main control thread doesn't need to wait on it. It may be useful to check if a process block is currently active IS_RUNNING (Labeled_process_block). An expansion on the use of CRITICAL is the idea of priority requests. A labeled process
is currently active IS_RUNNING (Labeled_process_block). An expansion on the use of CRITICAL is the idea of priority requests. A labeled process
block can be forked with a priority by saying DO A with PRIORITY:variable. Any CRITICAL blocks could arbitrate queued requests by levels of priority. An example of this mutual exclusion, based on priorities, is modeling a disk driver, which has levels of urgency depending on its data latency, requesting the UNIBUS, which is an asynchronous system.

To coordinate two or more separate control threads one could use some form of semaphore by setting and clearing a particular flag that is common to the control threads. Another mechanism that does not imply any implementation but clearly states what is desired is the SIGNAL(Labeled_Process_block,s) and RECEIVE(Labeled_Process_block,s). SIGNAL signals Labeled_Process_block with 's' and the control thread in the signaling block does not continue until the signal is RECEIVED. By using Labeled_process_blocks in both SIGNAL and RECEIVE we can explicitly choose who is communicating with whom.

- Block name [DESIGN]
- I/O List
  - Global declarations; (Design only)
    - Data Type definition
    - Constants
    - Universal Names
    - Procedures and Functions
    - MODULE definition
    - Macro definition
  - ENVIRONMENT Section; (Design only)
    - Power Consumption
    - Space Requirements
    - Thruput/Performance
    - Radiation Hardening
    - Reliability, Availability, Serviceability
    - Technology Selection(s)
    - Attribute Elaborations
  - BEHAVIOR Section;
    - Declarations
      - Data Types
      - Constants
      - Variables
      - Registers
      - Terminals
      - Clocks
      - Labels
      - External
    - Behavior Body
      - Assignments
      - Conditionals
      - Control

Figure 1

- Procedure Invocations
- Function Invocations
- Iterations
- Macro Elaborations
- Timing Constructs
- Sequence/Concurrence Constructs

Procedure Content
- Procedure name (argument list);
- Declarations
  - Data Types
  - Constants
  - Variables
  - Procedures and Functions
- Procedure Body
  - Assignments
  - Conditions
  - Controls
  - Procedure Invocations
  - Function Invocations
  - Iterations
  - Transfers

Function Content
- Function name (argument list):type_name;
- Argument list has only input arguments:
- Declarations (same as procedure)
- Function Body (same as procedure)

Figure 1 (cont.)

Macro Content

Macro name (argument list);
need a "procedural" macro expansion language, not limited to text insertion. Basically generates in-line (local) and appended (remote) code but may use full power of procedures in doing so.

Structure Section

Statements

Module Instantiations

MODULE Elaborations

TIME Section

Assertions (on signal timing relationships)

Label: expression

Label: (MIN, NORMAL, MAX)

Abstract I/O timing model

TEST Section (design only)

Functional tests

PERSONALIZATION Section

RAM, ROM, PLA content

Figure 1 (cont.)

References

Hoare 78 - Hoare, C. A. R., "Communicating Sequential Process," CACM, 21, 8, 1978, pp. 666-667.

WoodsH 81 Report of IDA Summer Study on Hardware Descriptive Languages, June 1-12, 1981.

Moffat 81 Moffat, D., "Enumerations in PASCAL, Ada, and Beyond," SIGPLAN Notices, Feb. 1981.

Enum 81 Correspondence concerning "Enumerations in PASCAL, Ada, and Beyond," in SIGPLAN Notices, May 1981.

Euclid 77 Lampson, B., J. Horning, R. London, J. Mitchell and G. Popek, "Report on the Programming Language Euclid," SIGPLAN Notices, February 1977.

PASCAL 75 Jensen, K., and N. Wirth, PASCAL User Manual and Report, 2nd edition, Springer-Verlag, New York, 1975.

Ada 80 Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.

Legard 77 Richard, F. and H. Legard, "A Reminder for Language Designers," SIGPLAN Notices, December 1977.

Wulf 73 Wulf, W. and M. Shaw, "Global Variables Considered Harmful," SIGPLAN Notices, February 1973.

APPENDIX B

APPLICABILITY OF ADA CONCEPTS TO HDL

APPENDIX B

APPLICABILITY OF Ada CONCEPTS TO HDL

An assumption of the group is that a new language, HDL, is being developed. Our charter was to evaluate concepts from Ada for applicability to HDL.

A common feeling among group members is that another fundamentally different approach should be evaluated by the HDL committee. That approach is to extend Ada for hardware design/specification by including one or more DoD supplied packages tailored to facilitate this. These packages would define design entities, components of them, terminals, signals of various kinds, clocks, other timing features, etc., and define functions pertinent to them.

The result of the group consists of the attached two tables, cross referencing each other. One is a list of concepts from the Workshop Report and the other is a list of concepts from the Ada Reference Manual.

The members of the Committee which prepared this Appendix were:

John Esch
Andrew Griffith
Keith Russell
Nick Mykris.

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code*
1.1	Documentation		
1.1.1	I/O Interface	P3	E
1.1.2	I/O Behavior	P8,S11,P1,T1	X
1.1.3	Behavior Decomposition	S11,P1,T1,P7, F3	X
1.1.4	Structural Decomposition	P4	E
1.1.5	Memory Content	T2,A4,I4	S
1.1.6	ASCII Character Set	A5	S
1.2	Design Oriented	S3,P1,L3	E
1.3	Use by Simulation (executable by a simulator)	C6	N
1.4	Use by Synthesis Tools	C6	N
1.4.1	Enumeration Type Representation	E4	S
1.5	Use by Testing Tools	?	N
1.6	Use by Physical Design Tools	?	N
1.7	Translation to Other HDL's	C6	N
1.8	Portability	A5	S
1.9	Application to Other Deliverable Items	?	N

* X ≡ Ada's is excessive, S ≡ Ada's is sufficient
E ≡ Ada's needs extension, N ≡ Ada does not have an applicable concept.

(Continued)

- LIST OF CONCEPTS -			Applicable Ada Concepts	Applicable Code
2.0	Hierarchy Considerations		P8,S11,P1,T1, P7,F3	X, E
	Characteristics of design models		"	X, E
	Composite system considerations		"	X, E
	Terminals/Ports		P3	E
	2.0.1	Entity Typing	P7	E
2.1	Reference of Objects		C2,N1,Q1	S
	2.1.0	Naming of Physical Entities	Q1	S
	2.1.1	Parameter Pairing	P3,B5	S
	2.1.2	Terminals or Ports	P3	E
	2.1.3	Records	R3	S
	2.1.4	Arrays	A4	S
	2.1.5	Attributes	A7	S
	2.1.6	Networks of Modules	P7	S,E
2.2	Collections of Related Entities		P1	X
2.3	Modularity		C6,S3,G1	E
	2.3.1	Interconnection of Entities	P3,P7,A6	E
2.4	Libraries		L3	S
	2.4.1	Iteration	C1,L7	E
2.5	Completeness		C6,S3	E
2.6	Recursion		R4	S (?)
3.0	Behavior		Seel.1.3,D2,B3	X,E
	3.0.1	Data Declarations	D2	X,E
	3.0.2	Program Control Statements	S4	X

(Continued)

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code
3.0.2.1	Repetition	L7	S
3 0.2.2	Selected	C1	S
3.0.2.3	Alternating	I2	S
3.0.2.4	Directed	G3	S
3.0.3	Transformations (Data Out is a function of Data In)	P1,P7	X,E
3.1	Behavior Description	P7	S
3.1.1	Abstract Data Types	P1	X,E
3.2	Data Types	T2	X,E
3.2.0.1	Define Data Types	T2,D2	S
3.2.0.2	Built-in Data Types	T2	E
3.2.0.2.1	Enumeration Types	E4	S
3.2.0.2.2	Integer Types	I5	S
3.2.0.2.3	Real Types	R2	S
3.2.0.2.4	Pointer Types	A1	S
3.2.0.3	Variable Range	R1	S
3.2.0.4	Grouping and Association of Objects	C8	S
3.2.0.5	Records	R3	S
3.2.0.6	Records with variant fields	V2	S
3.2.0.7	Data Precision	P5,A2	S

(Continued)

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code
3.2.0.8	Type Checking	T2	S
3.2.0.8.1	Overrideable	C7	S
3.2.0.9	Compiler defined constants for data storage attributes (Max_int)	I5, P1	S
3.2.1	Data Abstraction	P1	X,E
3.2.2	Enumeration	E4	S
3.2.3	Range	R1	S
3.2.4	Precision	P5,A2	S
3.2.5	User-defined Data Types	T2	E
3.2.5.1	Character Strings	C3	S
3.2.5.2	Bit Strings	B4,A4	S
3.2.5.3	Define New Data Types (records)	T2,S12	S
3.2.5.4	Pointers (Access variables)	A1	S
3.2 .5.5	Composite Data Types (records)	R3	S
3.2.6	Deferred Definition	I2,B5	S,E(?)
3.2.6.1	Translation to different levels of abstraction	P1	S
3.2.7	Strong Type Checking	T2	S
3.2.8	Coercions		
3.2.9	Reference to Attributes	A7	S
3.2.10	Scope and Visibility	S2,V3	X

(Continued)

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code
3.3	Operators and Expressions	A3,L6,E7,R5,B7	S
3.3.0	User definable operators and functions	O2	S(?)
3.3.1	Basic Operators	E7,C2,B1	S
3.3.2	Compare under mask	F3,O2	S
3.3.3	User-defined Operators	O2	S(?)
3.4	Control Flow	S4	E
3.4.0.1	Stepwise Refinement	P1,S11	S
3.4.0.2	Control Abstraction	T2,P1	E(?)
3.4.0.3	Parallel (concurrent execution)	T1,R7	E
3.4.0.4	Mutual Exclusion	T1,R7	E
3.4.1	Control Construction	S4	S
3.4.1.1	IF	I2	S
3.4.1.2	CASE	C1	S
3.4.1.3	LOOP	L7	S
3.4.1.4	EXIT	E6	S
3.4.1.5	BLOCK	B2	S
3.4.1.6	RETURN	R10	S
3.4.1.7	GO TO	G3	S
3.4.2	Control Abstraction	T2,P1	E(?)
3.5	Timing		N
3.5.0	Timing Delays (Nominal, Min,Max)	T2	S
3.5.1	Synchronous vs. Asynchronous		N
3.5.1.1	Interrupts	I6	?
3.5.1.2	Pipelining	T2,P1	E(?)
3.5.1.3	Finite State Machine	T2,P1	E(?)

(Continued)

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code
3.5.1.4	Concurrency	T1,R7	E
3.5.2	Simulation Option Interface	P1	S
3.5.2.1	Event Driven		
3.5.2.2	Clock Synchronous		
3.5.2.3	Path Tracing		
3.5.2.4	Packet Communication		
3.5.3	Timing Constraints	E7	E
3.5.3.1	Set-up Time, Hold Time, Pulse Width	T2,P1	S
3.5.4	Parallel Element Models (Synchronization-Rondezvous)	T1,R7	E
3.5.5	Rise and fall Times	T2,P1	S
3.5.5.1	Periodicity	T2,P1	S
3.5.5.2	Leading and Trailing Durations	T2,P1	S
3.5.5.3	Multistate Logic (HI, LOW, HIZ, UND)	T2,P1	S
3.5.5.4	Delays (NOMINAL, MINIMUM, MAXIMUM)	T2,P1	S
3.5.6	Clock Definitions	C4	E
3.5.7	Time Control Concepts	T1,R7	E
3.5.7.1	Concurrency	T1,R7	E
3.5.7.2	Path Delay	T1,R7	E
3.5.7.3	Race Condition	T1,R7	E
3.5.7.4	Arbitration	T1,R7	E
3.5.8	Termination of Currently Executing Procedures		N

(Continued)

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code
4.0	Structure	P7	E
4.0.1	Instantiation		N
4.0.2	Elaboration	E1	E
4.0.3	Default Parameters	D3	S
4.1	Connections		N
4.2	Generic Models	G1	S
4.3	Permutability	G1	S
4.4	Interaction in Structure Descriptions		N
4.5	Iteration in Structure Descriptions		N
4.6	Procedure Modeling	P7	E
4.7	Extensible Number of Hardware Primitives	L3,S3	S

(Continued)

- LIST OF CONCEPTS -		Applicable Ada Concepts	Applicable Code
4.8.1	Sets as a way to Model Networks		N
5.0	Guidelines	-	-
5.1	Kernel Plus Extensions Equals Language		N
5.2	High Level Parallelism		?
5.3	Verifiability		S
5.4	One Language		E
5.4.1	Economy in number of statement types		X
5.4.2	Friendly to the User		X
5.4.3	Extensibility through the use of generic components and operators	G1	S
5.4.4	Clarity and Visibility terseness		S
5.5	Technology Independence		S
5.6	Meaningful Key Words		S
5.7	Comments	C5	S
5.8	Compiler Directives	C7	S
5.9	Expression of Invariance	E7	E
5.10	LR(1) Grammar		S
5.10.1	BNF specification		S
5.11	Execution Conflicts		S
5.12	Multi-valued Logic	T2,P1	S
5.13	Incremental Compilation		N

- LIST OF CONCEPTS - (From Ada)	Reference Manual Section	Applicable Workshop Report Concepts (By Section)
A1 Access Type/Pointers	3.3	3.2.0.2.4, 3.2.5.4
A2 Accuracy Constraint	3.3, 3.5.6	3.2.0.7, 3.2.4
A3 Arithmetic Operators	4.4, 4.5	3.3
A4a Aggregates & their assignments	3.6.3	1.1.5
A4b Arrays	3.3	2.1.4, 3.2.5.2
A5 ASCII Character Set	3.5.2	1.1.6, 1.8
A6 Assignment	5.1	2.3.1
A7 Attributes	3.3	2.1.5, 3.2.9
B1 Base of Numbers	2.4.1	3.3
B2 Block	5.1	3.4.1.5
B3 Body	3.9	3.0
B4 Boolean Types	3.5.3	3.2.5.2
B5 Box	3.6	3.2.6, 2.1.1
C1 Case Statement	5.1	3.0.2.2, 2.4.1, 3.4.1.2
C2 Concatenation	2.6	3.3.1
C3 Character Type	3.5	3.2.5.1
C4 Clock (current time of day)	9.6	3.5.2
C5 Comment	2.7	5.7
C6 Compilation	10.1	1.3, 1.4, 1.7, 2.3, 2.5
C7 Compiler Commands	10.4	3.2.0.8.1, 5.8
C8 Composit Objects	3.6	3.2.0.4
C9 Constant	3.2	
C10 Context Specification	10.1	
D1 Decimal Numbers	2.4	
D2 Declaration	3.1	3.0, 3.0.1, 5.2.0.1
D3 Defaults	3.7	4.0.3
D4 Delay Statement	5.1	
D5 Delta of Reals	3.5.9	
D6 Discreet Types	3.5	
E1 Elaboration	3.1	4.0.2, 4.6
E2 END explicit	14.1	
E3 Entity	3.1	
E4 Enumerations	3.5.1	1.4.1, 3.2.0.2.1, 3.2.2
E5 Exceptions	11	5.14
E6 Exit Statement	5.1	3.4.1.4
E7 Expressions	4.5	3.3, 3.3.1, 3.3.5, 3.5.3, 5.9
F1 Fixed Point Numbers	3.5.9	

(Continued)

- LIST OF CONCEPTS - (From Ada)		Reference Manual Section	Applicable Workshop Report Concepts (By Section)
F2	Floating Point Numbers	3.5.7	
F3	Functions	6	1.1.3, 2.0, 3.3.2
G1	Generics	12	2.3, 2.4.1, 4.2, 5.4.3
G2	Global Variable	8.3	
G3	GO TO Statement	5.1	3.0.2.4, 3.4.1.7
I1	Identifiers	2.3	
I2	IF Statement	5.1	3.0.2.3, 3.4.1.1
I3	Incomplete Type Del	3.8	3.2.6
I4	Initialization	3.2	1.1.5
I5	Integers	3.5.4	3.2.0.2.2, 3.2.0.9
I6	Interrupts	13.5.1	3.5.1.1
L1	Lagels	5.1	
L2	Lexical Unit	2.2	
L3	Library Units	10.1	1.2, 2.4, 4.6
L4	Lines	14.3.2	
L5	Literals	2.4	
L6	Logical Operators	4.5	3.3
L7	Loops	5.5	3.0.2.1, 2.4.1, 3.4.1.3
M1	Modes (IN/OUT/INOUT)	6.1	
N1	Names	3.1	2.1
N2	Null	4.2	
O1	Objects	3.2	
O2	Operators/Overload Del	6.7	3.3.0, 3.3.2, 3.3.3
O3	Own Variables	7.3	
P1	Packages	7	1.1.2, 1.1.3, 1.2, 2.0, 2.2, 3.0.3, 3.1.1, 3.2.0.9, 3.2.1, 3.5.1.3, 3.5.1.2, 3.5.3.1, 3.2.6.1, 3.4.0.1, 3.4.0.2, 3.4.2, 3.5.5, 3.5.5.1, 3.5.5.2, 3.5.5.3, 3.5.5.4, 3.5.6, 5.12
P2	Paragraphing (Pretty Printing)	1.4	
P3	Parameters	6.1	1.1.1, 2.0, 2.1.1, 2.3.1
P4	Precedence	4.5	

(Continued)

- LIST OF CONCEPTS - (From Ada)	Reference Manual Section	Applicable Workshop Report Concepts (By Section)
P5 Precision	3.5.6	3.2.0.7, 3.2.4
P6 Primitive Type	7.2	
P7 Procedures	6	1.1.3, 2.0, 2.0.1, 2.3.1, 2.4.1, 3.0.3, 3.1, 4.0, 4.6
P8 Programs	10	1.1.2, 2.0, 2.1.6
Q1 Qualification	3.5.1	2.1, 2.1.0
R1 Range Constraint	3.3	3.2.0.3, 3.2.3
R2 Real Types	3.5	3.2.0.2.3
R3 Records	3.7	2.1.3, 3.2.0.5, 3.2.5
R4 Recursion	3.3	2.6
R5 Relational Operators	4.5.2	3.3
R6 Renaming (Aliases)	8.5	
R7 Rendezvous	9.5	3.4.0.3, 3.4.0.4, 3.5.1.4, 3.5.4, 3.5.7
R8 Representation Spec.	3.9	
R9 Reserved Words	2.9	
R10 RETURN Statement	5.1	3.4.1.6
S1 Scalar Type	3.5	
S2 Scope	8.2	2.1, 3.2.10
S3 Separate Compilation	10.1	1.2, 2.3, 2.5, 4.6
S4 Sequence of Statements	5.1	3.0.2, 3.4, 3.4.1
S5 Shared Variables	.11	
S6 Simple Statements	5.1	
S7 Slice (of arrays)	4.1	
S8 Spacing	2.1	
S9 Statements	5	
S10 Strings	2.6	
S11 Subprogram	6	1.1.2, 1.1.3, 2.0, 3.4.0.1
S12 Subtypes	3.3	3.2.5.3
S13 Synchronization	9.5	
T1 Tasking	9	1.1.2, 1.1.3, 2.0, 3.4.0.3, 3.4.0.4, 3.5.1.4, 3.5.4, 3.5.7
T2 Types	3.1	1.1.5, 3.2, 3.2.0.1, 3.2.0.2, 3.2.0.8, 3.2.0.2,

(Continued)

- LIST OF CONCEPTS - (From Ada)	Reference Manual Section	Applicable Workshop Report Concepts (By Section)
		3.2.5.3, 3.2.8, 3.4.0.2, 3.4.2, 3.5.0, 3.5.1.3, 3.5.1.2, 3.5.3.1, 3.5.5, 3.5.5.1, 3.5.5.2, 3.5.5.3, 5.12
V1 Variables	3.2	
V2 Variant Records	3.7	3.2.0.6
V3 Visibility Rules	8.1	3.2.10
W1 WAIT statement	9.7.1	

APPENDIX C

INDIVIDUAL COMMENTS

COMMENTS OF:

General Electric on the VHDL Description as contained in this Draft Report

Previous drafts of ths report have each contained a number of controversial areas in which two or more "options" were enumerated, and General Electric has commented and expressed argument in favor of particular statements in each case. We have participated in the evolution of the report into its present state, wherein all of these "options" have been resolved into a consensus, with many of our suggestions incorporated. General Electric congratulates the Woods Hole Committee on an excellent result, and endorses this report as a constructive step which will position the Tri-Service HDL Committee favorably towards its objectives.

COMMENTS OF:

R. Plesset--Rockwell International

In what follows I would like to state some of my thoughts on the important issues and as to why the goal of realizing an effective VHDL is a difficult one. A few recommendations will be made concerning attainment of this goal.

First, one has to come to grips with what is VHDL going to describe. "Hardware" has different connotations to different people as was evidenced to me at the workshop. Having been involved with the design of digital hardware systems and integrated circuits for a good many years I feel that hardware connotes boxes, cards, modules, ICs, resistors, etc., and relates to such considerations as physical, electrical, functional, environmental specifications, etc., especially as pertains to the military context. Software should be considered separately by reference to other language standards. The difficulty arises when the attempt is made to scope the range of behavioral description. The low end is more straightforward, e.g., circuit description where descriptive models have been around for a long time and selecting a standard descriptive vehicle seems relatively painless. However, if it is desired to impose syntax common to all levels of description, the selection or synthesis task becomes more complex. On the upper end is the boundary with system variables where descriptive level is abstract and hence not directly related to hardware. At this level important considerations are accuracies, cost effectiveness, reliability, maintainability, etc. Behavioral description is often mathematical and not concerned with implementation or "hardware". Again, imposing a common syntax between this level and other levels becomes difficult to accomplish, in addition to deciding where the translation to hardware should be. This leads to the question of what will

VHDL be used for. As I see it, the most meaningful applications would be (1) specification of hardware, (2) documentation of hardware database, (3) training and maintenance of hardware. Here specification is used to refer primarily in the procurement of new equipment. The problem here is to describe the desired equipment unambiguously without bias as to implementation and to convey enough detail to allow the reader to design or propose to design the equipment. Database documentation could be used in multiple sourcing or in using already developed hardware in alternate applications. Uses in training and maintenance are fairly obvious. A conflict arises where on one hand an HDL should be highly "readable" for uses involving people, but syntactically and semantically concise and precise for computer information systems on the other. A trade-off is required to be made to determine the optimum or practical mix. The use of the TI HDL as straw man provides an excellent reference for this trade-off. It is my belief that this HDL is the only one presently available that permits a "reasonably" complete coverage of hardware systems. I do feel, however, that its capability for describing behavior is somewhat limited. As I understand it, the intent by TI was to make this aspect of the language as general as possible in order to avoid a specification from implying or imposing implementation. I feel that this is desirable, but functional behavior should be permitted at lower levels. This could be accomplished by the use of user-generated ratio functions which could be called by the behavioral program.

In general, more flexibility should be given in relating structure and function where desired.

In conclusion, although I believe that a military standard VHDL is vital, "the language" will be slow in arriving. However, I recommend that the TI with some modifications be used in the interim. Further, I think that an HDL language as in natural languages is open-ended and, hence, must be evolutionary. To this extent, the TI HDL would improve with revisions. In addition, care should be given to restrict the scope of the language reference to "hardware", and leave system level description to those languages appropriate at that level. One possibility would be to modify the TI behavioral syntax to be comparable with Ada. This would promote better communication at least with people using Ada, which ultimately could be appreciable. The use of Ada syntax in VHDL should be limited to those semantics applicable to hardware.

With regard to simulation, I believe that a simulator should be built that will work with VHDL. The simulator should cover those aspects of behavior that can be described by VHDL. The problem of relating VHDL to other simulators has to be involved with translating it into specific industry design and development CAD systems.

With regard to the use of VHDL in design, I feel that whether it is useful in one or more aspects of design will depend on the individual user. But the emphasis should be on use of VHDL for description and specification.

COMMENTS OF:

Steve Piatz--Sperry-Univac

An important paragraph was deleted from draft 3 of the Report. "These recommendations do not reflect current DoD policy in general, nor of the VHSIC Program Office in particular, nor do they reflect the recommendations of the parent organizations of the participants of the summer study. The term VHSIC HDL should be read in that sense."

General Comments on Appendix A

1. The sample syntax used to describe concepts should not be taken as a binding on a VHDL.

2. The ability to describe designs with several alternate descriptions as shown in Fig. 2-1 on page 2-2 is not provided. The REDEFINES concept in TI-HDL is not adequate for this purpose.

3. The data abstraction concepts discussed have not been integrated into the language.

4. The Timing and Sequencing concepts discussed are not adequate for all design styles. This is perhaps due to the fact that our requirements are not well defined.

5. The TI-HDL allows multiple assignments in a single statement, for example:

   ```
   A:= B:= C;
   ```

   as well as concatenation on the left side of the assignment operator, for example:

   ```
   A:: B :=C,
   ```

   This syntax is error prone.

6. The TI-HDL allows identifiers (Names) that are enclosed in apostrophes to contain special characters, for example:

```
'123' := 100;
ABC   := '112';
```

In this example it is not clear if ABC is assigned the value 100 or the value 123. This becomes even more confusing as the needed abstract data types are added to the language.

Detailed Comments on Appendix A

1. Section A-1.2

   There appears to be no provision for alternate behaviors or structures as called for in Section 2 of the main report.

2. Section A-1.3

   The requirement that the top (or root) of the hierarchy be treated special (DESIGN VS MODULE) seems artificial. Since one person's system (DESIGN) is another person's component (MODULE), making a distinction between design and module seems unnecessary.

3. Section A-1.5

   It appears that this section should be separated into two sections--one dealing with globals in the behavior, and a second dealing with globals in the structure. This section should agree with the changes made to Section 3.2.10 of the main report. These changes require the explicit use of IMPORT and EXPORT constructs for globals in the VHDL.

4. Section A-1.7 and Section A-2.0
Having the terminals described within the BEHAVIOR section is only workable if there is only one BEHAVIOR section. The requirement is that there may be possibly many. Therefore, it is logical to place the terminal descriptions in the I/O interface section which is common to all the behaviors and structures. If this is not done, the VHDL user would have to enter a significant amount of data redundantly. The I/O Interface Section is implied in Section 2.0 of the main report.

5. Section 3.2.1.3 Guarded Commands
Since a behavior description prepared using guarded commands is allowed to be nondeterministic it seems that it may be desirable to provide an error detection mechanism. This mechanism could be analogous to the OTHERS clause in Ada, it would indicate the action to be taken if more than one guard is active simultaneously.

6. Section 3.2.1.3 Incremental Time & Schedule Statement
While the concepts expressed are not necessarily bad, they do not address all the design styles that were considered at Woods Hole. To be usable by a large variety of users, the concepts for time must be expanded to consider parallel or concurrent execution where different paths have different cumulative delays.

In addition, it should be indicated that, in the example syntax, the delay within the <> brackets can be a constant or an expression.

COMMENTS OF:

Prof. Fredrick J. Hill--University of Arizona

I read the report with interest particularly Sections 3, 4, and 5 even though I was not on hand during the second week of the Woods Hole Conference when this report was generated. These sections seem to form a sound specification for a language. It is interesting how similar the projected features of this language are to those of CONLAN. It seems a shame to repeat the agonizing process that went into the development of CONLAN.

With respect to the options cited in Sections 3, 4, and 5, I prefer the following:

Option 2 in 3.2.5
Option 2 in 3.2.10
Option 2 in 4.5
Option 1 in 5.7

In Section 5.11 an Option 3 which allows user (or toolmaker as in CONLAN) to define := and incorporate in his definition the appropriate result in the event of execution conflict would seem to be preferable to the listed options 1 and 2.

I do not think that the Starfish model for language extension is adequate. It is necessary to be able to exclude features from a user-level language as well as add to a kernel of low-level features. Certain types of operations necessary in the Kernel may not admit to a hardware interpretation at, for example, the RTL level. It will be necessary to provide for a formal exclusion mechanism if a user is to be provided with a language satisfying:

"If a feature is in the language it can be realized by hardware, and anything realizable in a particular hardware framework can be expressed in the language."

Apparently the critical decision to be made is whether to modify the TI language along the lines of WHDL or develop a new Ada-based VHDL. I agree that most of the important features of

VHDL have been accepted for inclusion in WHDL. Actually adding them to the TI language would be another matter. I fear that the result would be a patchwork which would satisfy no one. While it will obviously take longer, I recommend the formation of a paid working group to develop the ADA-based language consistent with Sections 3, 4, and 5. This working group should consist of qualified individuals from more than one organization. The complete CONLAN report will no doubt be available, when this group begins its work. I am certain that the CONLAN report will prove helpful, since the CONLAN objectives very nearly coincide with those set forth in the Woods Hole draft report.

COMMENTS OF:

Gary B. Goates--Boeing Aerospace Company

The TI HDL, with more or less modifications and extensions, appears to meet enough of the requirements defined in this document to be of considerable value to Phase I of the VHSIC program. Nevertheless, an optimal VHDL will require a fresh start. If such a new language development effort is pursued, how can it best be structured--taking into account what we know and don't know) about the craft of language design and implementation?

In my view, a new VHDL should not be "defined" by a "working group" of "experienced language designers", unless this group also has responsibility for implementing what they propose. This is because:

(a) A design committee without implementation responsibility will tend to compromise among participants' viewpoints by including additional constructs in the language. In contrast, a group that must both define and implement a language system will tend to select a small set of constructs that is sufficient to meet requirements, thus leading to a more elegant and more portable language.

(b) A language specification that exists only as text is a poor basis on which to evaluate the proposed language, just as a text description of a VHSIC component is much less useful than one that can drive a simulator.

(c) The process of implementing a language and validating it by processing nontrivial examples alters the language definition. In my experience, implementation reveals new language constructs and features that are useful and easily available. Validation reveals some missing features that were not recognized earlier as requirements.

I suggest that developing a new VHDL be divided into the following phases:

<u>Phase I - Requirements Definition</u>

I(a) Expressing, at least tentatively, what the VHSIC program requires of an HDL--i.e., this report.

I(b) Developing a consensus within the VHSIC community that this report is an adequate requirements definition, or identifying where it is lacking. What needs does the VHSIC program have that are not adequately covered here?

<u>Phase II - Alternatives Analysis</u>

II(a) Identifying several alternative languages, or approaches to designing and implementing a language, that meet the requirements identified.

II(b) Selecting two to four of these alternatives as "candidate" VHDLs.

II(c) Developing a prototype implementation and full specification of the candidate VHDLs including developing and processing the VHDL description of a DoD-specified test-vehicle circuit--with such implementation performed by the group that originally proposed the approach in Phase II(a).

II(d) Evaluating the candidate VHSIC HDLs and selecting one for production-quality implementation.

Phase III - Full-Scale Implementation

Phase IV - Transporting VHDL Support Software to the VHSIC Community

Phase V - Maintenance

This task breakdown is based on the precedent set by DoD in developing several experimental versions of the Ada language. It also follows the standard software engineering practice of conducting requirements definition and alternatives analysis phases before proceeding with full-scale implementation.

COMMENTS OF:

Gerry Marino--Raytheon Company

Two options were widely discussed for interpreting delay in a behavioral entity. These options are as follows:

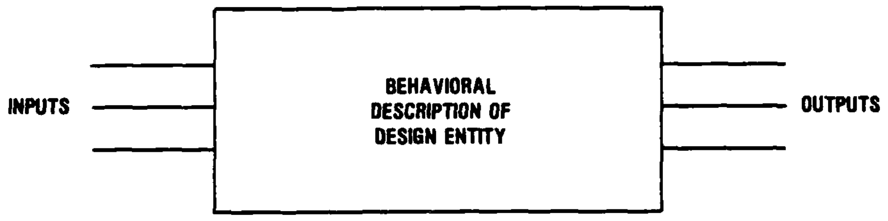
Option 1

The language should provide the ability to associate delays with all data transfers within a design entity. The value to be assigned is computed immediately but the actual assignment is not made until the delay has elapsed.

Option 2

All design entities should execute in zero time. When a delay is encountered in a signal path within a design entity that delay is added to the total propagation delay along that path. The total delay is reflected in the time delay associated with a signal change at the output of the design entity.

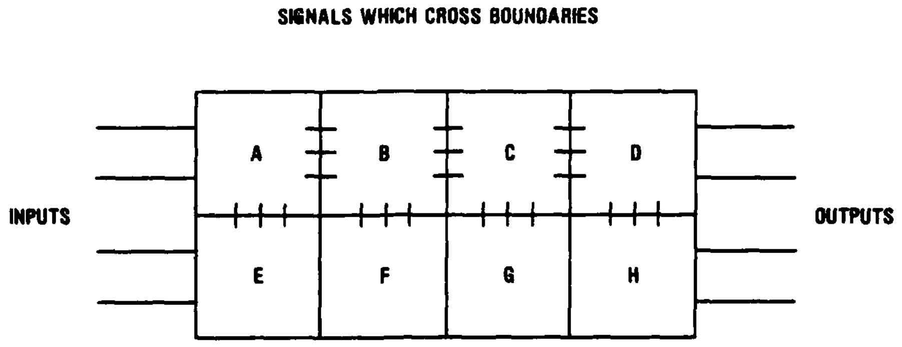
Option 2 describes the view of timing presently employed in the TI HDL. This view of timing in a behavioral design entity holds that each time the entity is evaluated, all data transfers are made immediately. Delays may be accumulated as evaluation proceeds through the various control sequences, but these delays actually take effect only at the output terminals. In my opinion this makes the concept of time for a behavioral model of a design meaningless. The following diagram represents a design entity at some level of the design hierarchy:



7-23-81-19

In working with this description, the user may only look at timing of signals from input to output. If, for example, there are two data items A and B in the entity, the user is unable to examine the timing of signals between the inputs and these internal items. The timing relationship between signals arriving at A and B may also not be examined. In other words, the user cannot work with a pure behavioral description, at any level of the design hierarchy, to examing timing relationships.

To add real timing, the user must create a structure composed of blocks. Each block may itself be a behavioral description. The point is that structure must be added to work with timing. The following figure shows a structural description of a design entity composed of several behavioral descriptions:



7-23-81-20

Timing may only be examined on signals which cross the boundaries of the behavioral blocks. The user cannot "open" a behavioral block and look inside to see internal timing. Also, when signals at block inputs change before previous signal changes have been reflected at the block outputs, the block may not react the way real hardware would. This is because the original signal results have already been transmitted to the outputs, regardless of internal delays, and are scheduled for transition on output terminals.

I feel that three categories of questions must be answered before the option 2 concept of timing may be used in a behavioral HDL.

(1) Is the concept of wall clock time (i.e., delay) not useful in a purely behavioral model for a process? In other words: do we want to enforce structure before a designer is allowed to consider delay time? If we enforce structure, will that impede the designer abstract thought process?

(2) When a structure is composed at behavioral block, is it adequate to examine timing only at the boundaries of the blocks, or will the user wish to "probe" timing within the behavioral blocks themselves?

(3) Given a structure composed of behavioral blocks, is the "instant" evaluation of a behavioral block adequate to model complex timing of signals arriving at random times at the block inputs? It must be noted that an HDL should be able to characterize the operation of a circuit under a wide range of conditions, including the application of unexpected sequences of input stimulus.

I feel that the concept of time presented in option 1 is more general and much more flexible. This allows delays to take place internally to behavioral models. Also note that option 2 is a subset of option 1. That is, if option 1 is accepted, the user may still lump his delay at the output terminals of his behavioral block.

The final judgment of which version of timing is to be used must lie with the end users of this language. For this reason, I feel strongly that various applications should be examined in the upcoming months.

COMMENTS OF:

N. Mykris--Rockwell

The scope of the purpose of a hardware description language has not been clearly defined. It is apparent that the workshop has tentatively decided upon the behavioral and structural aspects of hardware descriptions. However, the behavioral and structural organization aspects have been aimed at system level descriptions. No attempt has been made to concentrate on the HARDWARE related constructs of any HDL.

Although it is important to be able to describe a system abstractly without implications to hardware implementation, the development of systems through the various design methodologies is usually an iterative process of implementation strategies which reflect definite hardware structural components. There is an obvious transition from the system description to the hardware implementation; hence, the description of system designs should be decomposed into a behavioral description language and a hardware implementation language. This division is logical since the system designers are concerned with behavioral requirements and organization where the hardware designers are involved with implementation strategies. The relationships between the languages must be harmonious and isomorphic for complete information exchange and verification.

Ada, GPSS and other high-level languages are appropriate for the behavioral descriptions of systems. A hardware design language should reflect the structural implementation of the hardware realization. The hardware language should allow natural constructs for the various design methodologies. A hardware design language is useful only if the natural constructs are attractive from the hardware designer's viewpoint.

In conclusion, the scope of the hardware design language should be limited to the structural description of the hardware realizable constructs. No universal language is appropriate to cover the entire range of hardware description. The VHDL language should serve the purposes of design implementation description (hardware realization), hardware documentation, and hardware verification to the original behavioral specification.

COMMENTS OF:

Dr. Manuel d'Abreu--Honeywell

Myke Smith--Research Triangle Institute

We do not believe that the Ada approach is the best approach if we are to have an HDL for VHSIC Phase I. Ada was not intended to be an HDL nor has it ever been used as an HDL, and its body of experience is small and limited to use in the systems programming area. Ada is a more powerful language than the TI-HDL in the sense that it has more data types, more constructs, and incorporates some very high-level concepts. The question to ask is, are these all useful in a hardware description language? Many of these capabilities could be used in a "blue-box" description but are inappropriate at lower levels of description, for example, tasking and exception handlers. Looking at Ada from the hardware point of view, we see that Ada has no concept of concurrent statement execution, no concept of hardware delays, difficulty in expressing multiple instances of identical objects, and that the rendezvous concept is far removed from the hardware details. These are major shortcomings. Major reasons for using Ada are the big design effort, the DoD support, and the large software base. But, with the modifications to overcome the Ada shortcomings we will not have Ada! We will not have a DoD-supported Ada nor will we be able to use this software base. And, very importantly, the validation effort for Ada will be mostly for naught, given the modifications to Ada. We are then back to square one, and the alleged advantages of Ada are lost or at least considerably diminished.

We believe that the Woods Hole HDL is the best means for accomplishing a workable HDL within the VHSIC Phase I time frame. There are several reasons why we believe this is the better of the two approaches. The TI-HDL has been used for many years in a profit-making environment as an HDL and it has survived and flourished. Thus, the base language for the WHDL has a history of use as an HDL and a large body of experience. The suggested changes to the behavioral section are, in general, with the exception of guarded commands, well-known and well-understood programming language features.

The following is a series of tasks that we feel ought to be done to achieve an HDL based on the TI-HDL.

(1) The syntax and semantics of WHDL must be defined by a small team of experienced language designers. The WHDL should be based on the TI-HDL, a prioritized list of "desired attributes", and the stated intention that this language must be able to be simulated. There should also be a well-thought-out document that completely describes what the language is intended to do and what the language is not intended to do. The language designers should be given free rein to do the design. They should produce a syntax and semantics of an HDL. They should also produce a document specifying why any of the "desired attributes" were left out of the language. They should also show that they can describe the "desired attributes" with the language they present, in other words the language can do what we want in a clear and concise manner.

(2) A group of people familiar with HDLs should take this language and describe a highly parallel digital system, a highly pipelined digital system, a data flow based system, an object based system, and an I/O system such as the UNIBUS. These examples should be able to point out the shortcomings and weaknesses of the language. These problems can then be rectified if they are considered major problems.

(3) A validation effort for the language should be started in parallel with Tasks 1 and 2. This effort should be to produce a suite of descriptions that fully test the language's translator and provide a set of "test" descriptions with results to clarify any semantic problems. This document can then be used

as an implementor's guide for other versions of the translator and simulator. This would lessen the risk of HDL dialects.

Task 1 should take approximately 9-12 months and require about 3 man-years. This time will be heavily dependent on how much outside interference the designers have to suffer through. Task 2 should take about 3 to 4 man-months per project, assuming the projects were well-defined before they are started. We do not have enough experience in the validation area to hazard a guess for Task 3.

COMMENTS OF:

Committee to Clarify Hierarchical Terminology:

Chairman: Gary Goates--Boeing
Ernie Codier--GE
Steve Piatz--Sperry-Univac
Dick Plesset--Rockwell
Joel Seidman--Hughes

In discussing glossary entries, it became apparent that there was an intolerable variation in the use of the words "entity", "module", "block", and "component". A committee was formed to resolve the issue. The report was scanned for instances of words that attempted to refer to hierarchy. A quick survey of hierarchy in TI's HDL and in Ada was performed. A consensus emerged that two special terms (i.e., terms precisely defined and restricted in meaning) were required: one to refer to a node of a hierarchy *per se* and one to refer to a reference from a node to a lower node set that is included in its decomposition. A third perspective was used in some sections of the report--the perspective of a node looking toward its parent node (the larger system)--but it was decided that it was unnecessary to define a special term for this. The term "design entity" was tentatively adopted to refer to a node in and of itself. The term "components" was tentatively adopted to refer to the inclusion of a set of smaller design entities in a larger, or higher-level, design entity.

It is left to the reader to define any of the following terms that he or she may wish to use: assembly, block, conglomeration, design, element, field, intrinsic, instance, item, model, module, node, object, project, terminal, type, system, subsystem, and submodule.

COMMENTS OF:

Andrew D. Griffith--
Westinghouse Electric Corporation

The study has made it clear to me that what is needed in the long-range future is the growth to a true system description language that not only spans possible hardware designs but also spans hardware vs. software breakdowns. This language would allow a family of implementations from a flexible software implementation to a fixed hardware implementation.

COMMENTS BY:

Richard Rath--Hughes Aircraft Company

1.9 Translation to other HDLs

Automatic translation of high level languages is seen as a difficult theoretical problem. This is going to be one of the biggest obstacles to use of the VHDL as an information interchange standard. There is not going to be much value to documenting a VHSIC chip in computer-readable form if the computer at the receiving end cannot use it.

1.10 Portability

If the definition of portability given were applied to programming languages, they would all be portable. The second paragraph is not stated as a requirement of the language. In order that VHDL be portable it must be the case that many different CAD systems can generate and process it.

3.5.3 Timing Constraints

The concept of "level" applied to hierarchy is different than when used to distinguish between behavior description and component interconnects. This is a source of confusion.

5.1 Kernel Plus Extensions Equals Language

It is not appropriate to correlate the "level" of constructs provided with how widely those constructs are used or expected to be used.

5.3 Verifiability

This section appears to be addressing the problem of the completeness of a hardware description in VHDL. Verifiability should refer to the ability to evaluate the consistency between two distinct descriptions of the same system. Instead of requiring that a description be "executable" by a "simulator", the requirement should be that the semantics of the language be formally and unambiguously defined using a formal system for the expression of semantics. Phrasing the requirement this way leaves the way open for non-simulator-based verification techniques.

COMMENTS OF:

Dr. Sajjan G. Shiva, University of Alabama

Although I do not believe that the Ada Subset/Superset approach would be the way to VHSIC-HDL, it might be worthwhile to pursue that option along with the extensions to TI-HDL approach. This report can be a starting document for a small group of language designers. But it will be better if they have the inputs from the future language users. These inputs can be generated by the current VHSIC contractors through their efforts to describe some example systems in both of the above options.

Although a generalized (abstract) language approach is more elegant I feel it makes the language less usable (a common complaint on CONLAN).

The following group of comments pertains to a hypothetical patient monitor system:

PATIENT MONITOR FUNCTIONAL DESCRIPTION

This system is designed to utilize the data supplied from up to eight patient-monitoring instruments (blood pressure, heart beat rate, etc.), analyze the data, and report on the overall condition of the patient.

You may consider the system to have any architecture. For example, the instruments may be on one bus, or each instrument may have a separate serial or parallel line into the system, or the instruments may be "smart" and be part of the system itself. The outputs of the instruments may be analog or digital signals. They may be sampled or polled on demand.

The monitoring system is programmable. For example, a combination of results from several instruments may be combined arithmetically and a warning condition raised based on the result.

One of four conditions is output from the system: normal, warning, emergency, or instrument failure. In the case of failure, the system can be programmed to degrade "gracefully."

You should establish the overall design style, architecture, data rates, and instruction set, and attempt to model it on the TI HDL. Note any deficiencies in the language and recommendations based on the guidelines.

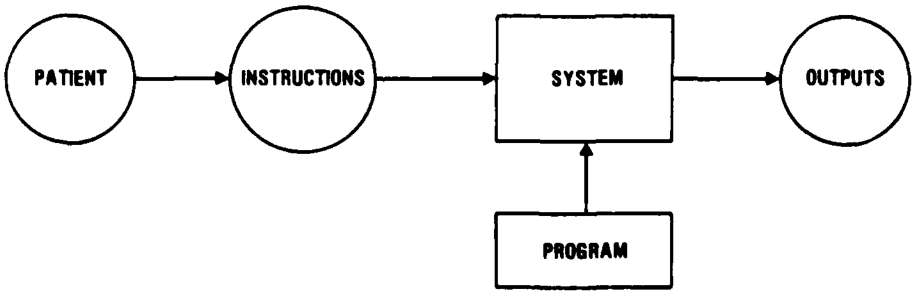
COMMENTS OF:
Menchini--INTEL; Piatz--Sperry-Univac;
Esch--Sperry-Univac; Franta--CDC;
Seidman--Hughes Aircraft;
Russell--National Semiconductor

PATIENT MONITORING SYSTEM

(8 units M_0 - M_7
each reports OK or FAIL)

```
S = P_COND = (OK, WARN, EMERG, FAIL);
P_STAT = RECORD
          E_Cond:  (OK, FAIL)
          CW_VAL:  VALUE
          INSTRUMENT:  STRING
          END;

M= ARRAY [ϕ..7] of E_STAT
P(M)[0...7]) P_COND
```

"P IS ONLY SPECIFIED TO ITS INPUTS AND OUTPUTS"



7-23-81-17

```
BLOCK S DESIGN;     (No type declarations)
     M (ϕ to 7) @ INPUT
     P          @ INPUT  are not Boolean
     O (ϕ to 3) @ INPUT

ENVIRONMENT
     TEMPERATURE OPERATION = 0 to 150
     VOLTAGES = 115/220 VAC @ 60/50 Hz  1 phase/3 phase
          (most not int TI HDL)

BEHAVIOR FUNCTIONAL PROGRAM  Not able to describe necessary
                             variables:
                             Need PASCAL description.

Must have implementation first.

CONST
     max-value = ?;
     min-value = ?;
     max-inst  = 8;

TYPE
     E_STAT = RECORD
              NAME: STRING:
              STATUS: (OK, DEAD)
              CURRENT_VALUE: min-value...max-value
     END
PROGRAM = RECORD
              UNKNOWN: ?
          END;

PATIENT_STAT = (OK, WARN, EMERG, FAIL);

EQUIP = ARRAY [O...MAX-INST] of E_STAT
```

BLOCK MONITOR DESIGN:

M: EQUIP @ INPUT

P: PROGRAM @ INPUT

C: PATIENT_STAT @ OUTPUT

ENVIRONMENT

VOLTAGE 115 VAC @ 60 Hz or 220 VAC @ 50 Hz, 3PH

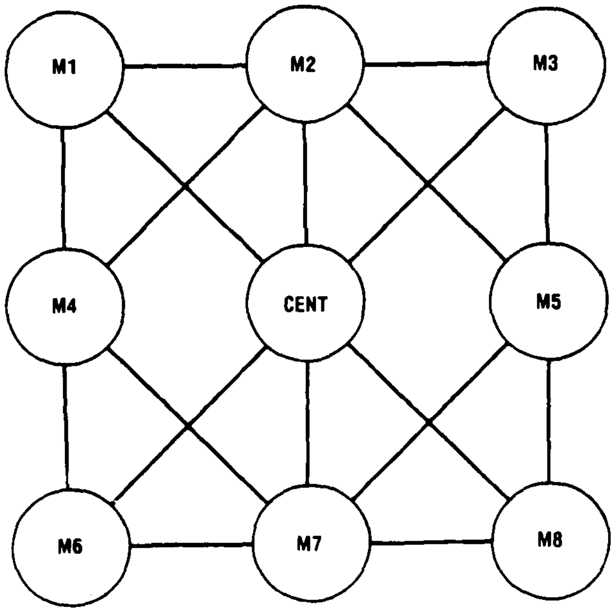
LEAKAGE 0.001 A MAX AT X VOLTS

TEMPERATURE 15 C to 30 C OPERATING AT 20 to 90 RHNC
0 C to 150 C STORAGE AT 0 to 95 RHNC

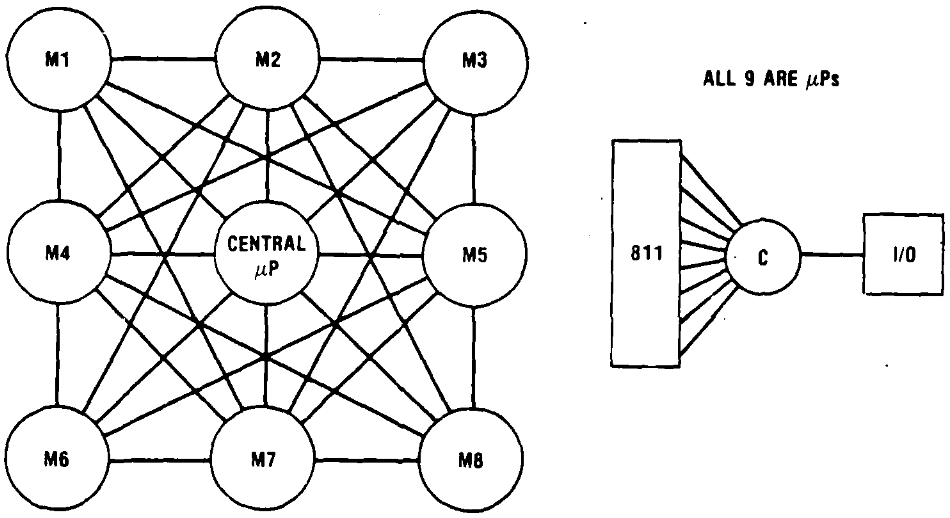
S = STATUS - comparison of machine value (M) to limits (LV or HV)

C^C = Patient Condition (normal, warn, emergency, fail)

```
Read (Limits: HV (0,7), LV (0,7))

1 DO 20 from x=0 to x=7
     If M(x)<LV(X) go to 5 else
     If M(x)>HV(X) go to 10 else
          s(X) = 00
          Go to 20

      5 S(X) = 01
          Go to 20
     10 S(X) = 10
     20 end

Read (Patient Subroutine, S,(0,7))
     "Different for each patient"
Programmed by Doctor--Interactive
     I/O set into EPROM.
```



7-23-81-18



*Easy to see relationship between C and any M.

*Interaction between M_x and M_y

1. Priority Interrupt

   Each machine sets % of time necessary for reporting based on "activity" and "sensitivity".

2. Sending of warning or emergency code to C based on its own or collection of other inputs compared to its own.

*Each M has "signature" and sends/receives information from C, and other M.

7-23-81-21

COMMENTS OF:
    C. Leung--MIT

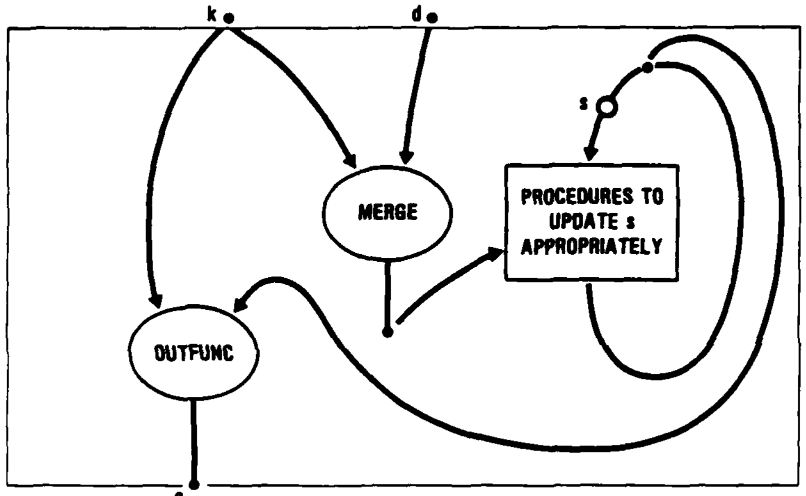
A PACKET SYSTEM DESIGN FOR THE PATIENT MONITOR PROBLEM

Design to be refined: written in ADL

```
patient monitor:  module (param: n-of-instruments)
                  inlet p:  new-patient-data, /* information
                      about a new patient */
                  d:  instrument-readings (n-of-instruments) /*
                      instantiation of type instrument-readings
                      with parameter n-of-instruments */
                  outlet o:  monitor-patient-condition /*
                      instrument failure, critical, etc. */

behavior
     state s:  pm-state: = initial-pm-state; /* pm-state
               determines which instrument reading to ignore,
               data about individual patients, his blood
               pressure, pulse rate, etc. */
     m: monitor shares s /* provides an orderly way to enter
                         data about new patient, new status
                         about various instruments attached
                         to the patient currently under
                         monitoring, etc. */

     new patient procedure (x: new-patient data);

     update:  s: = x;
     end new patient;
```

```
          new-instrument-status-procedure (y: instrument readings
                                              (n-of-instruments));
          update s: = new-instrument-stat (s,y);
          end new-instrument-status;

     end m;
     m. new-patient (p); /* enter data for a new patient */
     m. new-instrument-status (d); /* new set of instrument
                                      readings */
     0   outfunc (d,s); /* determine output signal from current
                           state and new readings*,
     end; patient monitor
```

(NOT A DATA FLOW GRAPH!!)



PATIENT MONITOR

DATA ABSTRACTIONS TO BE REFINED:

(Data representation + operations for manipulating these representations)

- new-patient-data
- instrument-readings with parameter n-of-instruments
- monitor-patient-condition
- pm-state

PROCEDURAL ABSTRACTIONS TO BE REFINED:

(Procedures in terms of control structures such as conditions and looks, other data abstractions and operations defined for data abstractions)

- new-instrument-stat: pm-state x instrument readings — pm-state
- outfunc: instrument-readings x pm-state — monitor-patient-condition

OTHER ISSUES

- how to transform physical signals into packets
- can be done in TI/HDL by giving implementation for handshake protocol and using the structure facilities extensively

7-23-81-3

COMMENTS OF:

Lionel Bening--Control Data Corp.

```
BLOCK PATIENT_MONITOR
DIN(1 TO 8)  @INPUT
DIW(1 TO 8)  @INPUT
DIE(1 TO 8)  @INPUT
DIF(1 TO 8)  @INPUT

DON(1)       @OUTPUT
 DOW(1)      @OUTPUT
 DOE(1)      @OUTPUT
 DOF(1)      @OUTPUT

BEHAVIOR FUNCTIONAL PROGRAM

     IF(DIF(1 TO 8)   B'00000000') THEN
         DOF(1) = B'1'
         SCHEDULE DOF (1)
     ELSE
         DOF(1) = B'0'
         SCHEDULE DOF (1)
     ENDIF
     IF (DIE(1 TO 8)   B'00000000') THEN
         DOE(1) = B'1'
         DOW(1) = B'0'
         DON(1) = B'0'
     ELSE
         IF(DIW(1 TO 8)   B'00000000') THEN
             DOE(1) = B'0'
             DOW(1) = B'1'
             DON(1) = B'0'
```

```
          ELSE
               DOE(1) = B'0'
               DOW(1) = B'0'
               DON(1) = B'1'
          ENDIF
     ENDIF
     SCHEDULE DOE(1)
     SCHEDULE DOW(1)
     SCHEDULE DON(1)
     EXIT
END PATIENT_MONITOR
```

COMMENTS OF:

R. Waxman--IBM/FSD

DECISION TABLE

CONDITION	NORMAL RANGE	X		X		X		X		X		X		X		X
	WARNING		X	X			X	X			X	X			X	X
	EMERGENCY				X	X	X	X					X	X	X	X
	INSTRUMENT FAILURE								X	X	X	X	X	X	X	X
OUTPUT INDICATOR	NORMAL LIGHT	X								X						
	WARNING LIGHT		X	X							X	X				
	EMERGENCY LIGHT				X	X	X	X					X	X	X	X
	INSTRUMENT FAILURE LIGHT								X	X	X	X	X	X	X	X

7-23-81-22

COMMENTS OF:

Scott E. Perkins--Fairchild

APPENDIX

Outline:

1. The "Strawman" versus the MPPMS
2. Scoping
3. Axiomatic Systems Design
4. Alternative Architectural Styles
5. System Architecture for the MPPMS
6. The MPPMS Described in TI-HDL

1. THE "STRAWMAN" VERSUS THE MPPMS

I had an opportunity to put the TI-HDL "Strawman" to the test. I designed an MPPMS--a Multi-Processor Patient Monitoring System--and then coded up the design in the TI language (at the end of this comment you will find an MPPMS system diagram and a copy of the TI-HDL description of the system). The TI-HDL showed itself to be better, from a programming standpoint, than the RTL (Register Transfer Language); however, the exercise also suggested areas where the TI-HDL could be improved. The coding was made unnecessarily less <u>structured</u>, less <u>modular</u>, and more time-consuming because the TI-HDL does not have <u>Procedure Declarations</u>, or <u>User-Defined Data Types</u>. My experience suggested the following principle:

"A modern HDL should be constructed as a <u>superset</u> of a structured, high-level programming language."

An application of this principle to the TI-HDL suggests to me that the language could greatly benefit from the work done at Stanford on an HDL compiler, called ADLIB. ADLIB adheres strictly to a Pascal syntax. I believe the TI-HDL could and should be modified so as to include Pascal or Algol-68 as a subset.

2. SCOPING

My experience in using the TI-HDL also brought to my attention the problem of <u>scoping of variables</u>, <u>user-defined constants</u>, <u>user-defined types</u> and <u>procedures</u>. For example: should there exist global variables, or should variables be passed only as arguments of functions and procedures? And since many languages handle these constructs in the same way, how should VHDL handle these constructs? It seems to me that, as we design a VHDL compiler, we should be at least as consistent in these matters as the designers of Pascal.

3. I found Allen Razdow's presentation, "Introduction to Applications of HOS to Hardware Design," to be one of the most

innovative of the HDL concepts presented at this conference. To put it simply, Allen's concept is to use a very high-level language called AXES to describe the functionality of a hardware system in an axiomatic way. The goal is to be able to describe a system in such a way as to not force the designer into one particular architectural style or another. I personally feel that the HOS concept represents the only legitimate proposal made so far for dealing with complex systems involving networks of processors.

4. ALTERNATIVE ARCHITECTURAL STYLES

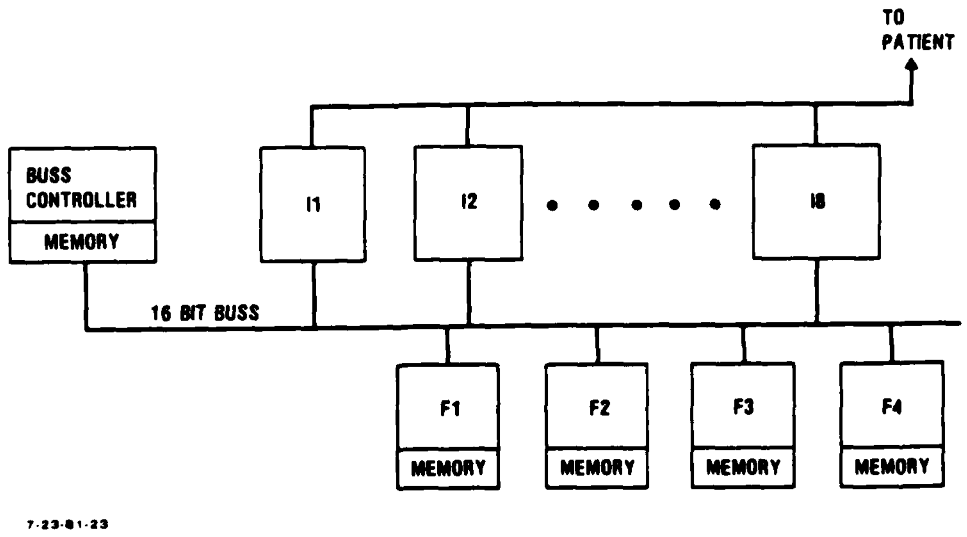
Dr. Clement Leung, from MIT, and Dr. Suhas Patil presented excellent reports on why the traditional RT (Register Transfer) will soon be yielding to alternative architectures, such as:

(a) SLAs (Storage Logic Arrays)

(b) Data Flow Architectures.

Hardware Design Languages must be flexible enough to allow people to design using different architectural styles.

5. THE SYSTEM ARCHITECTURE OF THE MPPMS



```
Block MPPMS Design                        (*By Scott E. Perkins*)
(*A Multi-Processor Patient Monitoring System*)

(*Described using the Texas Instruments Hardware Description
Language*)

Buss (0 to 15) Output;
Structure
(*Buss Controller Micro-Processor*)
BC:  Buss_Controller Buss (0 to 15);
(*Instruments*)
I1:  Instrument Buss (0 to 15);
I2:  Instrument Buss (0 to 15);
I3:  Instrument Buss (0 to 15);
I4:  Instrument Buss (0 to 15);
I5:  Instrument Buss (0 to 15);
I6:  Instrument Buss (0 to 15);
I7:  Instrument Buss (0 to 15);
I8:  Instrument Buss (0 to 15);
(*Function Processors*)
F1:  Function Buss (0 to 15);
F2:  Function Buss (0 to 15);
F3:  Function Buss (0 to 15);
F4:  Function Buss (0 to 15);
(*end structure*)
(*Notes on MPPMS Buss Structure*)
(*
 DB:  bits 0-7  (Data Bus)
 AB:  bits 8-11 (Address Bus)
IOF:  bits 12   (Instrument Output Flag)
FIF:  bit 13    (Function Input Flag)
FOF:  bit 14    (Function Output Flag)
FBF:  bit 15    (Function Begin Flag)
*)
```

```
(*Begin Functional Program*)
Behavior Functional Program;
(*Initialize Buss*)
Integer I, J, K, L, M, Time;
I: = 0; J: = 0; K: = 0; L: = 0; M: = 0; Time: = 0;
for I: = 1 to 15 do Buss (I): = 0;
(*Address sequentially the eight different instrument processors.
Then, for each instrument, put the data for instrument j and
put it onto the data buss.  Then using the address buss, address
the four function processors sequentially, instructing them to
stack the information from the data buss into RAM inside each
of the function processors.  To do this we will use a nested FOR
loop structure where the outer loop is indexed with a J indicating
the Jth instrument and the inner FOR loop is indexed with a K,
indicating the Kth function processor*)
(*The case statement is used to generate the address of the
instrument processors*)
FOR j: = 1 to 8 DO
Begin FOR L: 8 to 11 Do Buss (L:): = 0;
     Case j of
     1:  Buss (8): = 0;
     2:  Buss (8): = 1;
     3:  Begin Buss (8): = 0; Buss (9): = 1 end;
     4:  Begin Buss (8): = 1; Buss (9): = 1 end;
     5:  Begin Buss (8): = 0; Buss (9): = 0; Buss (10): = 1 end;
     6:  Begin Buss (8): = 1; Buss (9): = 0; Buss (10): = 1 end;
     7:  Begin Buss (8): = 0; Buss (9): = 1; Buss (10): = 1 end;
     8:  Begin Buss (8): = 1; Buss (9): = 1; Buss (10): = 1 end;
END case;
(*For each generation of an Instrument Address
increment time by 100*)
Time: = time + 100;
(*Put address on Address Buss*)
Schedule Buss at time;
```

```
(*Set IOf: = 1, IOF is bit 12*)
Buss (12): = 1;
Schedule buss at (time + 20);
(*Reset Instrument Output Flag to 0*)
Buss (12): = 0;
Schedule Buss at (time + 40);
(*By (time + 60) we should have the data from the Jth instrument
out onto the data buss so let us try to now read it into the four
function processors.  We will first set up the addressing code
for the function processors.
Let us reset the address buss to 0*)
FOR K: = 1 to 4 DO
Begin for M: = 9 to 11 DO Buss (M): = 0
    Case K of
    1:  Buss (11): = 1; (*0001*)
    2:  Begin Buss (8): = 1; Buss (11): = 1 end;
    (*1001*)
    3:  Begin Buss (8): = 0; Buss (9): = 1; Buss (11): = 1
    end; (*0101*)
    4:  Begin Buss (8): = 1; Buss (9): = 1; Buss (11): = 1
    end; (1101*)
End Case;
(*Now we have a function address on the address buss.  Let us
put up the function input flag at time +60+ (K@*4), so that the
buss is scheduled at time +60 +4, 8, 12, 16.*)
Buss(13): = 1;
Schedule Buss at (time +60+ (K@*4));
(*After two units of time we can go ahead and shut off the
function input flag*)
Buss (13): = 0;
Schedule Buss at (time +60+ (K@*4) + 2);
END (*Function Loop Compound Statement*)
END; (*Instrument Loop Compound Statement*)
```

```
(*At time: = 900 units we should have all the instrument readings
stored in RAM inside the function processors.  It is now time
to begin processing the information in the function processors.
Let's start by setting time: = 900 and doing the addressing*)
time: = time + 900;
FOR K: = 1 to 4 DO
Begin
     FOR M: = 8 to 11 Do Buss (M): = 0;
     Case K of
     1.  Buss (11): = 1; (*0001*)
     2.  Begin Buss (8): = 1; Buss (11): = 1 end; (*1001*)
     3.  Begin Buss (8): = 0; Buss (9): = 1; Buss (11): = 1
     end; (*0101*)
     4.  Begin Buss (8): = 1; Buss (9): = 1; Buss (11): = 1
     end; (*1101*)
     end case;
(*Let's now start processing function K*)
(*Set Function Begin Flag to 1*)
Buss (15): = 1;
Schedule Buss at (time + K@*20);
(*This means that the functions will begin executing every 20
units*)
(*Then we now shut off the FBF*)
Buss (15): = 0;
Schedule Buss at (time + (K@*20) + 5;
end (*Compound Statement of Function Begin loop*)
(*Let's assume that at 2000 units the function processors have
finished analyzing the data from the Instruments.  Now we must
interrogate the function processors.  So let's go through the
addressing business again*)
Time: = time + 2000;
FOR K: = 1 to 4 Do
```

```
Begin
     FOR M: = 8 to 11 Do Buss M: = 0;
     Case K of
     1:  Buss (11): = 1; (*0001*)
     2:  Begin Buss (8): = 1; Buss (11): = 1 end;
     (*1001*)
     3:  Begin Buss (8): = 0; Buss (9): = 1;
     Buss (11): = 1 end;
     (*0101*)
     4:  Begin Buss (8): = 1; Buss (9): = 1;
     Buss (11): = 1 end; (*1101*)
(*Set Function Output Flag to 1*
Buss (14): = 1;
Schedule Buss at (time +K@*20);
(*Then shut off FOB five units later*)
Buss (14): = 0;
Schedule Buss at (time +(K@*20)+5);
end; (*end Function Output Compound Statement loop*)
(*At time = 3000 units the Buss controller microprocessor has
had time to look at the outputs of the function processors and
schedules output on the data buss which tells the physician if
the patient is healthy.*)
```

APPENDIX D

SPEAKERS AND PRESENTATIONS

APPENDIX D

SPEAKERS AND PRESENTATIONS

1. Dave Ackley -- "TI HDL"
2. Lionel Bening -- "CDC Automated Integrated Design System (AIDS)"
3. John Esch and Steve Piatz -- "Sperry-Univac HDL Spec."
4. Gary Goates -- (a) "Storage/Logic Arrays"
   (b) "ABLE: A Layout Modeling Language"
5. Fred Hill -- "A Hardware Programming Language"
6. Clement Leung -- "Data Flow/Architecture Description Language"
7. Willie Lim -- "Hierarchical and Iterative Structure Description Language"
8. Leon Maissel -- "Interactive Design Language"
9. Paul Menchini -- "The LCM Chip Design Methodology"
10. Hillel Ofek -- "Language for Computer Design"
11. Suhas Patil -- "The Element of Style in Digital System Design"
12. Allen Razdow -- "High Order System for Hardware Design"
13. Sajjan Shiva -- "Hardware Synthesis from an HDL Description"

APPENDIX E

PARTICIPANTS IN IDA SUMMER STUDY ON HARDWARE DESCRIPTION LANGUAGE

APPENDIX E

PARTICIPANTS IN IDA SUMMER STUDY ON HARDWARE DESCRIPTION LANGUAGE

Coordinator: Glenn W. Preston
Institute for Defense Analyses
400 Army-Navy Drive
Arlington, VA 22202
(703) 558-1749

Mr. David Ackley
Texas Instruments
P.O. Box 226015
Dallas, TX 75266
(214) 343-7888

Mr. Lionel C. Bening, Jr.
Control Data Corp.
4201 Lexington Avenue North
Arden Hills, MN 55112
(612) 482-2215

Mr. Ernest Codier
General Electric Company
Aerospace Electronics Systems Dept., MD 348
French Road
Utica, NY 13503
(315) 797-1000 x7939

Dr. Manuel d'Abreu
Honeywell, Inc.
12001 St. Hwy #55
Plymouth, MN 55441
(612) 541-2303

Dr. John W. Esch
Sperry-Univac
Univac Park
P.O. Box 3525
St. Paul, MN 55165
(612) 456-2222

Mr. Paul M. Franta
Control Data Corp.
4201 Lexington Avenue North
M/S ARH 251
Arden Hills, MN 55112
(612)482-2326

Mr. Glenn Glatfelter
The Mitre Corp.
Bedford, MA 01730
(617) 271-2000

Mr. Gary Goates
Boeing Aerospace
P.O. Box 3999, M/S 88-07
Seattle, WA 98124
(206) 773-1038

Mr. Andrew Griffith
Westinghouse Electric Corp.
P.O. Box 746, M/S 451
Baltimore, MD 21203
(301) 765-6322

Dr. Fredrick H. Hill
Dept. of EE, Building #20
The University of Arizona
Tucson, AZ 85721
(602) 626-5136

Dr. Clement K.C. Leung
Laboratory for Computer Science
Massachusetts Institute of
Technology
545 Technology Square, Room 538
Cambridge, MA 02139
(617) 253-6012

Mr. Willie Lim
Laboratory for Computer Science
Massachusetts Institute of
Technology
545 Technology Square, Room 532
Cambridge, MA 02139
(617) 253-6035

Dr. Leon Maissel
IBM Corporation
Dept. C-14, Building 704
Boardman Road
Poughkeepsie, NY 12602
(914) 463-2302

Mr. Gerry Marino
Raytheon Company
Missile Systems Division
Hartwell Road
Bedford, MA 01730
(617) 274-7100

Mr. Paul Menchini
INTEL Corp.
M/S AL 3-2-471
3585 SW 198th Avenue
Aloha, OR 97005
(503) 642-6282

Mr. Nick M. Mykris
Rockwell Intl. Corporation
Avionics Group Headquarters
D/288 MC 124-411
400 Collins Road, NE
Cedar Rapids, IA 52406
(319) 395-4664

Mr. Dan Nash
Raytheon Company
Missile Systems Division GRA-4
Hartwell Road
Bedford, MA 01730
(617) 274-7100

Mr. Hillel Ofek
IBM Corporation
Building 951-3, Dept. B-22
P.O. Box 950
Poughkeepsie, NY 12602

Prof. Suhas Patil
Patil Systems
391 Chipeta Way, Suite C
Salt Lake City, UT 84108
(801) 581-4482

Mr. Scott Perkins
Fairchild R&D
4001 Miranda Avenue
Palo Alto, CA 94304
(415) 483-3100 x2351

Mr. Steve Piatz
Sperry-Univac
Univac Park, P.O. Box 3525
St. Paul, MN 55165
(612) 456-2364

Mr. Richard Plesset
Rockwell Intl. Corporation
D/549 MC 136-BD24
P.O. Box 4761
Anaheim, CA 92803
(714) 632-3490

Mr. Richard Rath
Hughes Aircraft Company
P.O. Box 902, E51-A278
El Segundo, CA 90245
(213) 616-2303

Capt. J.B. Rawlings
AFWAL Avionics Laboratory
Wright Patterson AFB
Ohio 45433
(513) 255-6553

Mr. Allen Razdow
Higher Order Software, Inc.
806 Massachusetts Avenue
Cambridge, MA 02139
(617) 661-8900

Mr. Keith Russell
National Semiconductor, D-3660
2900 Semiconductor Drive
Santa Clara, CA 95051
(408) 737-3410

Mr. Richard G. Sanders
The Aerospace Corporation
P.O. Box 92957
Bldg. A3, Room 2293
Los Angeles, CA 90009
(213) 648-5000

Mr. Nick Schmitz
General Electric Co.
ELAB
Building 3, Room 116
Electronic Park
Syracuse, NY 13221
(315) 456-2826

Mr. Joel Seidman
Hughes Aircraft Company
P.O. Box 902, E51-A278
El Segundo, CA 90245
(213) 616-2083

Dr. Sajjan Shiva
Computer Science Department
University of Alabama
P.O. Box 1247
Huntsville, AL 35899
(205) 895-6160

Mr. Myke Smith
Research Triangle Institute
Building #5
P.O. Box 12194
RTP, NC 27709
(919) 541-6829

Mr. Bill Stewart
Honeywell Information Systems
P.O. Box 6000, M/S B104
Phoenix, AX 85005
(602) 866-3030

Mr. Ron Waxman
IBM Federal Systems Division
9500 Godwin Drive
Manassas, VA 22110
(703) 367-2167

APPENDIX F

GLOSSARY

GLOSSARY

Abstraction

An abstraction is an alternative way of looking at a complex system or process that is in some way simpler and easier to deal with. It is characterized by the fact that it avoids the details of the complex system, modeling them as simpler concepts. The value of an abstraction is that it allows the complex system to be manipulated and its internal and external interactions to be understood without the burden of considering all the details of the system or systems. A simple example of an abstraction is the notion of a hardware register as an abstraction of an assemblage of flip-flops. One can then speak of storing an integer number in a register, without worrying about binary representations. Because an abstraction hides complexity, it may be an imperfect model of the detailed system. This means that conclusions based on abstractions may be invalid when compared with a detailed view of the system.

ADL

Artwork Description Language (elsewhere the acronym signifies Architecture Description Language).

Behavior

Specifies a design entity in terms of the functional and timing relationships between the input and output ports of the network. Behavior describes the function of a design entity as opposed to its composition. It tells what a network does rather than how it is built.

Cell

A chip structural entity describing physical implementation and related functional attributes.

Coercion

A mechanism for mapping information between two dissimilar information types.

Component

An element in the decomposition of a design entity. A component may itself be subject to further decomposition if it is subsequently viewed as a design entity. If it is not subject to further decomposition, it is a primitive component (q.v.).

Concurrency

Two or more simultaneously occurring sequences of events proceeding independently.

Control Abstraction

A concept that groups a sequence of substates into a single state, or considers a complex series of operations as a single operation, for the purpose of raising the level of abstraction of the design description.

Data Abstraction

A concept (taken from software engineering) that includes both grouping data objects (e.g., fields) into higher-level data objects (e.g., records) and defining specialized higher-level operations on data objects.

Data Object

A unit of data which can be used as an operand in behavioral descriptions.

Declaration Section

That part of the behavior description containing identification of the local variables of interest and the signals used to communicate to the other parts of the system.

Design Entity

In a graphical representation of a design hierarchy, each vertex denotes a design entity. Less formally, a design entity is a unit in the structural hierarchy; it is well defined in terms of its input/output ports and its behavior. A design entity is by definition decomposable into components. A design entity may also be viewed (looking downward in the hierarchy) as a component in a higher-order design entity.

Entity

Any named object. This is the broad application of the term as found in Ada. In this document, the term design entity (q.v.) is used in a narrower sense.

Execution Section

That part of the behavior description containing program control statements and expressions detailing the data transformations expected in the system described.

Generic Instantiation (Elaboration)

The process of defining a design entity from a generic object by specifying user-supplied parameters for the generic object. These parameters can be used to set up default conditions for a subsequent component instantiation process.

HDL

A Hardware Description Language (see Preface).

Incremental Compilation

An attribute of a particular implementation of a language system that allows language statements to be acted upon (either compiled or interpreted) statement by statement.

Instantiation

The process of specifying the remaining object parameters, supplying terminal interconnections, and a particular name for an occurrence of an object type.

Library

A collection of related software or hardware entities grouped together for easy access by computer processes. In this document, libraries are mandated to be coded in VHDL syntax.

Parallelism

Two or more simultaneously-occurring sequences of events interacting at various times through interdependent data exchange or handshaking, resulting in interdependent processing.

PLA

A Programmable Logic Array. A rectangular array of AND and OR gates for generating a group of functions in sum-of products form.

Primitive

A design entity in the lowest hierarchical level, which is not decomposed further, but is completely characterized by a behavioral description in the VHDL.

Primitive Component

A component which is not subject to decomposition, and is described only by a behavioral description.

Procedural Model

A procedural model of a structural description is a procedure that upon execution returns a structural description. Thus, the structural description can be dependent on parameters passed to the model as well as on global variables.

Self-timed

A hardware system is self-timed if operations of its modules are synchronized by using ready/acknowledge handshake protocols locally, and not by referencing a global timing signal.

SLA (Storage/Logic Array) Program

A two-dimensional array of symbols (taken from an SLA cell alphabet) that encodes both the functionality and the topology of a circuit design.

Structure

Specifies a design entity in terms of modules and their interconnections. Structure tells how it is built rather than what it does.

TDL

A Test Description Language.

Terminal

An external connection point for a signal.

Type Checking

A mechanism for insuring consistent usage and interpretation of data objects between various parts of a behavior or structure.

Windowing

An attribute of a particular implementation of a design system that allows the user to control the level of detail shown at a particular time of hierarchical aspects of design entities (e.g., depth of structural decomposition into components, level of data or control abstraction, etc.).

DATE
ILMED

-8